

Remote File Access System for Generic Ericsson Processor Boards

DANIEL JESÚS GARCÍA MORAL



**KTH Information and
Communication Technology**

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden



ROYAL INSTITUTE
OF TECHNOLOGY

Remote File Access System for Generic Ericsson Processor Boards

File transfer service, Random Access Memory-based file system and secure file transfer
solution research

DANIEL JESÚS GARCÍA MORAL

Master's Degree Project
Supervisor: Lukas Karlsson
Examiner: Mark Smith

Stockholm, Sweden October 2011

Abstract

Generic Ericsson Processor boards are general purpose hardware platforms which provide generic processing services. They support the Unified Extensible Firmware Interface Specification. They have several network interfaces available and they are connected to Ericsson's laboratory network. Several servers are also connected to this network.

These boards require periodic firmware upgrades. They also require acquiring new firmware components and data files. Currently, an application to download or upload files from and to Ericsson's laboratory servers when an Operating System has not already been booted does not exist. Therefore, the files have to be transferred to USB drives which are connected later to the boards in order to transfer the files. This is a time consuming operation which decreases Ericsson's productivity. In addition, although Generic Ericsson Processor boards have an optional solid-state drive as secondary storage, Ericsson wants to be able to operate without it. This is because this secondary storage is not always available and Ericsson does not want to use it when the Generic Ericsson Processor boards are operating before an Operating System has been loaded. They prefer to use Random Access Memory storage.

This project is focused on studying possible solutions for those two problems. Several file transfer protocols are analyzed. Several file system solutions mounted on Random Access Memory are also explored. A Trivial File Transfer Protocol client application and a Random Access Memory Disk driver prototype are designed, implemented and tested. They are tailored to work on a pre-boot environment, when the boards have not booted an Operating System yet, in Ericsson's laboratory network.

Finally, a secure file transfer protocols' study is developed. This study will be used to assess Ericsson on the optimal secure file transfer protocol choice in order to implement possible secure future versions of the system.

Keywords UEFI, BIOS, TFTP, FTP, UEFI Shell, RAM Disk, GEP boards, firmware applications

Acknowledgments

I will start with Lukas Karlsson, my supervisor. It has been his guidance and expertise which have made this Master Thesis work possible. In addition, he has also shown a lot of sympathy and he has been really supportive, specially when I was facing what seemed a dead end. Thank you Lukas.

I profoundly appreciate the help and support from the rest of the team: Thomas Ekstrand, Magnus Johansson, Erik Hamren and Per Fogelström.

I am also grateful to all to those who have been working like me on their own master thesis work and who understood the inner pressure which can be felt during this process. I am talking about you: Irena, Marko, Marcus, Isaac, Manuel, Ignacio and Armando.

Finally, I would like to thank Thomas Vennberg for giving me this great opportunity and Mark Smith, my examiner, because he has always been there when I needed him.

Stockholm, October 2011

Daniel Jesús García Moral

Contents

Contents	v
List of Figures	x
List of Tables	xii
1 Introduction	1
1.1 Problem description	1
1.2 Goals	2
1.3 Purpose	2
1.4 Target audience	3
1.5 Methodology	3
1.6 Limitations	4
1.7 Background	4
1.7.1 BIOS & UEFI	4
1.7.2 File transfer protocols	6
1.7.3 RAM disk drives	8
1.8 Project outline	9
2 UEFI	10
2.1 UEFI specification version 2.1	10
2.1.1 Introduction	10
2.1.2 Design overview	11
2.1.3 UEFI Images	13
2.1.4 Protocols	15
2.1.5 How UEFI drivers are implemented on GEP BIOS	16
2.2 The UEFI Shell	17
2.3 EDK	19
2.3.1 Intel Platform Innovation Framework for UEFI	19
2.3.2 The EDK project	19
2.4 Setting up the EDK environment	19
2.4.1 EDK Requirements	19
2.4.2 Equipment	20
2.4.3 Installing the EDK	21

2.4.4	Running a 32-bit emulator	21
2.5	Writing, building and testing UEFI Shell applications	22
2.5.1	UEFI Shell applications' components	22
2.5.2	How to build external UEFI applications	23
3	File transfer client application prototype	25
3.1	Requirements and state of the art	25
3.1.1	Requirements	25
3.1.2	Existing solutions	26
3.1.3	Available services to build a file transfer client application . .	27
3.2	First FTP prototype using the EFI Toolkit	27
3.2.1	Adaptation to the GEP environment	28
3.2.2	Problems with the first prototype	30
3.3	Final TFTP Prototype Design	30
3.3.1	Selecting the file transfer protocol to be implemented	30
3.3.2	The TFTP protocol	33
3.3.3	TFTP Option Extensions	36
3.3.4	Design choices - MTFTP4 protocol	38
3.4	Final TFTP Prototype implementation	40
3.4.1	Structure	40
3.4.2	Overview	43
3.4.3	Initialization	45
3.4.4	Get file operation	51
3.4.5	Put file operation	54
3.4.6	Help operation and TFTP error handling	55
3.4.7	Helper functions	56
3.4.8	Difficulties	57
3.4.9	TFTP server application choice	58
3.5	Analysis & Evaluation	60
3.5.1	Command line arguments' tests	61
3.5.2	TFTP-related tests	64
3.5.3	Timeout and try count tests	66
3.5.4	Limit tests	66
3.5.5	Block size tests	68
3.5.6	Throughput tests	70
3.6	Conclusions	74
3.7	Future work	74
4	RAM Disk drive prototype	76
4.1	Motivation and background	76
4.1.1	Motivation	76
4.1.2	RAM Disk drives	76
4.2	Requirements and state of the art	78
4.2.1	Requirements	78

4.2.2	Existing solutions	78
4.2.3	Available services to build a RAM Disk drive	79
4.3	First prototype using the EFI Toolkit	80
4.3.1	Adaptations to load the drive while the BIOS is being loaded	80
4.4	Final RAM Disk driver prototype design	84
4.4.1	Design choices - Disk file system	84
4.4.2	Design choices - Cluster size	85
4.4.3	Design choices - FAT16 boot sector	85
4.4.4	Design choices - BLOCK_IO protocol	86
4.5	Final RAM Disk driver prototype implementation	88
4.5.1	Differences of the final prototype with the EFI Toolkit's driver	88
4.5.2	Structure	89
4.5.3	Code data structures	91
4.5.4	Loading process	93
4.5.5	Read blocks operation	98
4.5.6	Write blocks operation	99
4.5.7	Flush blocks operation	100
4.5.8	External parameters handler functions	100
4.5.9	EFI Toolkit ported helper functions	101
4.5.10	Using the RAM Disk information to access the RAM Disk when the OS has been booted	102
4.5.11	Changing the RAM Disk device name	103
4.5.12	Difficulties	104
4.6	Analysis & Evaluation	105
4.6.1	Command line arguments' tests	105
4.6.2	Limit tests	107
4.6.3	Other tests	109
4.7	Conclusions	111
4.8	Future work	111
5	Secure file transfer protocols study	113
5.1	Motivation	113
5.2	TLS/SSL & SSH	113
5.2.1	TLS/SSL	113
5.2.2	SSH	114
5.2.3	Comparison	114
5.3	Secure file transfer protocols	115
5.3.1	FTPS	115
5.3.2	SFTP	116
5.3.3	SCP	117
5.3.4	Discarded options	118
5.4	Existing resources	119
5.4.1	UEFI 2.1 capabilities already existing in the BIOS	119
5.4.2	EDK2 resources	120

5.4.3	Porting other resources	121
5.5	Comparison	122
5.5.1	Discussion	122
5.5.2	Conclusion	125
5.5.3	Future work	125
Glossary, Bibliography & Appendices		127
Glossary		127
Bibliography		133
Appendices		139
A Block sizes measures' data		140
B Throughput measures' data		145
C GEP board's architecture		148
D TFTP client functions summary		150
D.1	Functions written in the TFTPClient file	151
D.1.1	TFTPClientMain()	151
D.1.2	TFTPClientGetLineHelp()	151
D.2	Functions written in the tftp file	151
D.2.1	tftpHandle()	151
D.2.2	tftpConfigure()	151
D.2.3	tftpLaunch()	151
D.2.4	tftpGet()	151
D.2.5	tftpGetFileSize()	152
D.2.6	tftpPut()	152
D.2.7	tftpCheckPacket()	152
D.3	Functions written in the helpers file	153
D.3.1	UtoA10()	153
D.3.2	StrToIpT()	153
D.3.3	UnicodeToAsciiT()	153
D.3.4	existingFile()	153
D.3.5	writeFile()	153
D.3.6	readFile()	153
D.3.7	openFile()	153
D.3.8	EfiAsciiStrnCmp()	153
D.3.9	AtoU64()	153
E RAM Disk drive functions summary		154

E.1	Data structures	154
E.1.1	Data structures were already written in the EFI Toolkit's source code	154
E.1.2	Data structure written by the author of this thesis	155
E.2	Functions	155
E.2.1	Core operation functions	155
E.2.2	Helper functions	157

List of Figures

1.1	EFI position related to the OS and the firmware	5
1.2	UEFI Specification evolution	6
1.3	Project structure	9
2.1	UEFI Overview.	12
2.2	UEFI Booting sequence.	13
2.3	Construction of a protocol.	16
2.4	UEFI Shell appearance	18
2.5	UEFI shell applications' files	23
3.1	RRQ/WRQ Packet	34
3.2	DATA Packet	34
3.3	ACK Packet	35
3.4	ERROR Packet	35
3.5	Request packet with options	36
3.6	OACK Packet	36
3.7	TFTP client structure	41
3.8	TFTP client overview	43
3.9	TFTP client initialization	45
3.10	TFTP client get file operation	51
3.11	TFTP client put file operation	54
3.12	TFTP get line help operation	55
3.13	Average download throughput with variance	69
3.14	Average upload throughput with variance	69
3.15	Average throughputs comparison with variance	72
3.16	Download throughput density function	73
3.17	Upload throughput density function	73
4.1	Configuring the driver's loading process	83
4.2	RAM Disk drive structure	90
4.3	RAM Disk loading process I	93
4.4	RAM Disk loading process II	94
4.5	Read blocks operation	98
4.6	Write blocks operation	99

List of Figures	xi
4.7 Flush blocks operation	100
C.1 GEP board's architecture	149

List of Tables

3.1	Command line arguments tests I	62
3.2	Command line arguments tests II	63
3.3	TFTP-related tests	65
3.4	Limit tests	67
3.5	Results	72
4.1	Shell's file commands tests	106
4.2	Limit tests	108
4.3	Other tests	110
5.1	Comparison	124
A.1	Time to download a 10-megabyte file in milliseconds for different block sizes	141
A.2	Time to upload a 10-megabyte file in milliseconds for different block sizes	142
A.3	Average download throughput in megabytes per second for several block sizes	143
A.4	Average upload throughput in megabytes per second for several block sizes	144
B.1	50 Download throughput measures in megabytes per second using default parameters	146
B.2	50 Upload throughput measures in megabytes per second using default parameters	147

Chapter 1

Introduction

This chapter describes the master thesis. It also introduces the reader to the subject of *Basic Input Output System (BIOS)* firmware, *Unified Extensible Firmware Interface (UEFI)*, file transfer protocols and *RAM* disk drives. First, the problem description is introduced. Second, the goals and purpose of the thesis are presented. The target audience is described. Then, the methodology adopted to solve the problem and the work's scope are outlined. Then, the background of the thesis is explained. Finally, the project's outline explains how the chapters are constructed and related to each other.

1.1 Problem description

This master thesis will investigate the solution to two problems stated by Ericsson:

- Remote access to files in *Generic Ericsson Processor boards* (GEP boards): The GEP boards which are used in Ericsson's laboratory often need to upgrade their pre-boot software and firmware (software and firmware which will be used when an *Operating System (OS)* has not already been loaded). There is also a need to get new tools and applications. In addition, the *Read Only Memory (ROM)* of these boards may not be able to store all the software needed at pre-boot time in future versions of the boards' firmware. The current solution consists on using USB drives to move the applications and updates from the laboratory servers to the GEP boards' RAM. It is a time-consuming operation which should be replaced by a more suitable system which does not need physical access to any machine apart from the board which is going to get or put files on the servers. This system could be used to perform automatic tasks like periodic software and firmware updates.
- Secure file transfer study: Ericsson's laboratory is a secure environment which can use a non-secure file transfer protocol like *FTP* or *TFTP* (which are

explained in section 1.7) to perform file transfers. Nevertheless those protocols cannot be used in an open environment where security is critical. Currently there is no need for a secure file transfer system to be run on GEP boards. Nevertheless, it has been foreseen that the need for this system could arise in the future. Therefore a study which investigates the possible implementations of such a system must be performed.

A description of GEP boards can be found on Appendix C.

1.2 Goals

- Create a Remote File Access System to be used on GEP boards on Ericsson's laboratory environment. This system will be loaded and used during pre-boot time, when there is no OS running on the boards yet. It will be run on top of the boards' pre-boot environment which supports *UEFI* specification version 2.1 (UEFI and the UEFI specification are introduced in section 1.7). It will allow downloading files from the laboratory servers and uploading files to them. The system will be automatically configured when the boards start running. It will be the foundation of automatic mechanisms designed to upgrade the boards and get software, firmware and data from the network. This system will be composed by two different artifacts: a non-secure file transfer client application and a *RAM disk* drive. These systems have never been implemented on this type of boards yet.
- Write a user's manual which describes how to use the Remote File Access System. It will be an Ericsson's private and internal user manual whose target audience will be Ericsson's employees which want to use the Remote File Access System. It describes how to configure and use this system, as shown by [39].
- Write design specifications which describe how the Remote File Access System was designed and implemented, in order to be able to maintain it and upgrade it if required, as shown by [40] and [38].
- Perform a study which investigates the possible implementations of a secure file transfer system on GEP boards.

1.3 Purpose

The purpose of this report is to describe the design and implementation of a Remote File Access System used to transfer files between Ericsson's laboratory servers and GEP boards. This system eases software and firmware upgrades and new software

and firmware installation, avoiding the need to physically access the servers. It also serves as a foundation to create an automatic system which will be able to load software and firmware from Ericsson's laboratory network on GEP board's RAM. Therefore, GEP boards can be booted using only the most basic firmware routines stored in their ROM. Once the platform is booted the Remote File Access System can be used to download applications and drivers from Ericsson's servers if needed. Consequently, Ericsson's productivity will be increased due to the faster automatic procedures, which will be able to be built using this system, in order to upgrade and acquire new software and firmware faster. It will also avoid increasing GEP boards' ROM size by loading the data from the network on the boards' RAM.

It also describes a study investigating the possible design and implementation of a secure file transfer system on GEP boards. This system could be used directly by Ericsson. It would increase the amount of servers that the boards would be able to communicate with. Therefore, it would ease upgrading and acquiring software, firmware and data files from external networks.

1.4 Target audience

The main target audience of this master thesis is people interested in developing software for platforms which support the UEFI specification. It can be specially interesting for those developing network applications or dealing with RAM-based file systems on computer pre-boot environments. This master thesis will be also interesting for Ericsson GEP users and developers.

Finally, researchers and students willing to have an understanding of pre-boot software development and UEFI specification current status and future directions may also find this work useful and interesting.

1.5 Methodology

In order to fulfill the aims and goals of the master thesis, qualitative and quantitative approaches were used. The employed qualitative method was secondary research, performing a literature review which was necessary to achieve a solid knowledge foundation. This literature review allowed to understand the current state of the art and to be aware of the software that was already available to be used to create new applications and systems. It also helped to understand the advantages and disadvantages of different file transfer protocols and RAM disk solutions. This was necessary to make choices concerning the file transfer client application and RAM Disk drive to be implemented.

Subsequently, these prototypes were implemented. A quantitative approach was employed to test the prototypes. Several test cases were designed and executed

to evaluate them. These test cases aimed to analyze and test the quality of the parameters which Ericsson demanded. The tested data was statistically analyzed and processed.

Finally, a research on how to implement a secure file transfer protocol on GEP boards was carried out. To analyze the feasibility of the possible solutions, secondary research techniques were employed. These techniques allowed to elucidate the requirements of each existing protocol and to discover the facilities which were available at GEP boards in order to implement them.

1.6 Limitations

First of all it was hard to conduct a detailed literature review. The UEFI forum (which created the UEFI specification) was created in 2005 and the UEFI specification is still on a relative early phase. The available documentation is enough for developing simple systems and applications. Nevertheless, it is scarce for developing more complex systems and applications. Consequently, it was usually necessary to examine the GEP boards BIOS' source code to understand the available services and to use them as a model for developing new software.

There was also a time limit constraint to perform the master thesis work: 20 weeks. Due to this time constraint it was not possible to implement a more complex secure file transfer protocol. Nevertheless security was not needed by Ericsson because the system is used on a secure environment.

Implementing the prototypes for GEP boards also entailed limitations. The prototypes have only been tested on a GEP environment. In addition, the hardware and software limitations of these boards, specifically the addressable memory, have conditioned the prototypes' design and usability.

1.7 Background

1.7.1 BIOS & UEFI

BIOS

Inside every computer there is a BIOS. A BIOS is a type of software which connects computers' hardware with the OS. It is the first program executed by the processor and it introduces the *Central Processing Unit (CPU)* to the mother board's basic components. Its main task is to prepare the computer environment in order to select and load an OS, as shown by [65].

The term was first used by Gary Kildall and first appeared in the *Control Program for Microcomputers (CP/M)* OS in 1975. It was used to describe the part of the

machine which was loaded during boot time and which interfaced with the hardware, as shown by [4]. First types of BIOS were stored in ROM or *PROM* chips. Therefore they could not be altered by users. Due to computer science evolution, the need for reprogrammable BIOS appeared in the early 1990s. BIOS firmware started to be stored on *Electrically Erasable Programmable Read Only Memory (EEPROM)* or *flash memory* devices. They also started to take more complex functions which include, among others, power and thermal management[4].

During the mid-1990s the first Intel-HP *Itanium* system development started. There were some *PC* BIOS limitations, namely the 16-bit processor mode, only 1MB addressable memory space and *PC AT* hardware dependencies which were unacceptable. To solve these problems the Intel Boot Initiative was created. It was later renamed to *EFI*, as explained by [32].

EFI

EFI was an improved replacement of the old legacy BIOS firmware interface which was used by all *IBM PC compatible personal computers* [32]. It redefined the BIOS concept: EFI is a software interface between an operating system and platform firmware as shown by Fig 1.1.

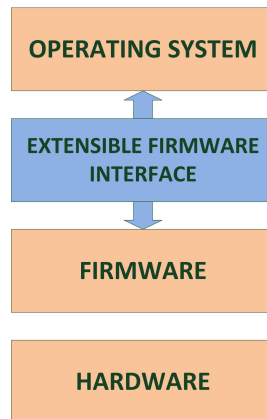


Figure 1.1: EFI position related to the OS and the firmware

This interface consists of several artifacts: data tables that contain platform-related information; boot services and runtime services (which are available to the OS and its loader). Therefore it offers a standard environment for booting an OS. It also allows running pre-boot applications, as stated by [18].

In 2000 EFI released its first specification, EFI version 1.02. EFI version 1.10 was released in 2002, and it included an EFI driver model. In 2005, the *Unified EFI* Forum was created. It is an industry-wide organization which promotes the adoption and continue the development of the EFI Specification (which was renamed to UEFI Specification), as shown by [18].

UEFI

The UEFI Forum released version 2.1 of the UEFI specification on January 2007. Improved cryptography, network authentication and a *Human Interface Infrastructure* (HII) were among its improvements. The UEFI Specification continued to improve. Its last version is UEFI v2.3.1, released in 2011, as shown by Figure 1.2.



Figure 1.2: UEFI Specification evolution

The Remote File Access System is designed to run on GEP boards. GEP boards comply with UEFI version 2.1 specification. As it was stated before, EFI and UEFI software interfaces can be used to boot an OS and they also allow running pre-boot applications. The Remote File Access System works on this pre-boot environment in which there is not an OS running.

1.7.2 File transfer protocols

File Transfer Protocol

FTP was created by Abhay Brushan on April 1971. When it was designed *Transmission Control Protocol (TCP)* and *Internet Protocol (IP)* did not exist yet. The first version was described in *Request For Comments (RFC)* 114 [8]. The specification evolved and RFC 765 was released in 1980. The current specification is RFC 959 [48], which was released in 1985.

FTP is intended to promote sharing of files and transfer data reliably and efficiently, as stated by [48]. It operates on top of TCP, in the TCP/IP network architecture. The file transfer process requires a server waiting for incoming requests on TCP port number 21. This connection is called control connection. It is used to manage the session and it involves commands' handling and identification and password administration, as shown by [29]. The control connection is open during all the session, as shown by [11]. There is a second connection used to transmit the files which is called data connection.

FTP has a login feature which allows restricting access on a user name/password basis. Therefore the server can give different privileges to each user. There exists a FTP simpler version called *Trivial FTP* which is widely used in *Preboot Execution Environments* (PXE). The Remote File Access System is run in a PXE.

Trivial FTP

In 1980 TFTP was defined in an *Internet Experiment Note (IEN)* by Sollins [53]. The current version is defined in RFC 1350, which corrected the *Sorcerer's Appren-*

tice Syndrome problem, as it is stated in RFC 1350 [52]. The Sorcerer's Apprentice Syndrome made the TFTP protocol useless, because all the packets in a TFTP transfer were infinitely replicated, consuming all the usable bandwidth and creating network congestion levels which were not acceptable.

It is implemented on top of *User Datagram Protocol (UDP)* and it uses port number 69 [52]. It is not a session oriented protocol like FTP. All the file transfers start with a request to read or write a file. If the TFTP implementation follows [52] without extensions, the file is divided and sent in fixed blocks of 512 bytes when it is transmitted. This limitation can be overcome if the TFTP block size option extension is used, following RFC 2348 [33]. Therefore the limit can be increased to 65464 bytes.

It uses timeouts and acknowledgments to support reliable transfer of files, due to the fact that UDP is not reliable, as opposed to TCP. It defines three modes of transfer:

- Netascii: A modified form of ASCII which is described in [47].
- Octet: Used to transfer arbitrary 8-bit bytes.
- Mail: Using the email address of the recipient as the file name it allows to send files to an email address. It is obsolete [52].

TFTP has several disadvantages compared to FTP. The most important are:

- It does not provide authentication, therefore it is only safe to use it on private, local environments.
- It cannot list directories.
- It is less user-friendly than FTP.

Nevertheless it also has some advantages:

- It is really simple and does not need a big amount of memory to be implemented.
- It is useful for booting devices which do not have non-volatile storage devices available. The *Bootstrap Protocol* is usually used in conjunction with the TFTP protocol to configure this type of devices when a system is starting

up. The Bootstrap Protocol is a network protocol whose function is to deliver IP addresses from a configuration server to network clients. It is defined in RFC 951 [10]. Nonetheless, GEP boards already have a network-based boot system. Consequently, this advantage is not critical for the project.

All these advantages are essential when developing a file transfer service for GEP's pre-boot environment as it will be explained later.

Security flaws

In May 1999, RFC 2577 was released. In this RFC several FTP security flaws are enumerated [1]. Among all the problems, the most critical one is the absence of encryption: user names, passwords, commands and data are sent in clear text. Therefore anyone could capture this information on the network and use it to access private information and impersonate users. TFTP is even less secure than FTP. In addition to its lack of encryption it does not support any authentication mechanism. These security flaws are not a problem when the file transfer service is run on a secure environment. On the other hand, if they are going to be used in open, non-secure environments they are critical.

1.7.3 RAM disk drives

Although GEP boards do have a solid-state drive which can be used as *disk storage* or *secondary storage* it is an optional device which is not used in all GEP boards, as it is shown in Appendix C. In addition, Ericsson prefers to use RAM memory rather than secondary storage when the GEP boards are operating before an OS has been loaded (in the pre-boot environment). Consequently, there is a need to implement a file system on a RAM disk in order to develop a working Remote File Access System. A RAM disk or RAM disk drive is a portion of RAM configured to simulate a *disk drive*. It is used to access files on the RAM the same way they would be accessed on a *physical drive*. They are faster than a physical drive. The information is stored in RAM, therefore the contents are no longer available after a system reset, as explained by [46].

Jerry Karlin invented and wrote the first RAM disk software for *microcomputers*, named *Silicon Disk System*, as stated in [6]. The motivation was enabling early microcomputers to use more RAM than the CPU could directly access. Hard drives were not readily available at that time and RAM disk drives were faster.

In 1983 Microsoft added a RAM disk to its *MS-DOS* OS. *AmigaOS* followed this initiative in 1986. Apple also added this software to their *Apple Macintosh* in 1991. AmigaOS still supports this feature in its 4.1 version. Apple *Mac OS X* has some utilities to create, format and mount RAM disks, as shown by [49]. RAM disk software has also evolved. It is still used in Microsoft *Windows OS* and there

are many commercial solutions, as shown by [27]. They are also used in UNIX systems, like Linux. Nevertheless, *tmpfs* is a much more common solution in UNIX environments, which consists on storing files on virtual memory maintained by the UNIX *kernel*, as shown by [51].

A RAM disk is a very good solution for the GEP Remote File Access System. Being faster than physical drives is a great advantage. Being volatile is not a disadvantage because the Remote File Access System is used to handle files which will not be needed after a system reboot.

1.8 Project outline

Figure 1.3 depicts the report's structure. The description of each chapter can be found below.

Chapter 1, Introduction Gives some background information and describes the problem, the goals, the purpose, the target audience and the methodology of the work.

Chapter 2, UEFI Presents deeper background information on UEFI: what is it, why should it be used, how is it structured, how can its protocols and services be used.

Chapter 3, File transfer protocol prototype Compares FTP and TFTP protocols and presents deeper information about TFTP structure and messages. It also describes the design, implementation and analysis and evaluation of a TFTP client application prototype.

Chapter 4, RAM Disk drive prototype Gives deeper information about RAM drives and explains design choices, implementation, analysis and evaluation of a RAM disk drive prototype.

Chapter 5, Secure file transfer protocol study Analyzes several secure file transfer protocols.

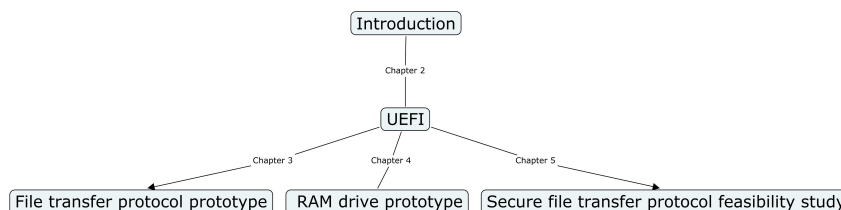


Figure 1.3: Project structure

Chapter 2

UEFI

This chapter aims to provide the reader some information about the GEP platform environment. Many concepts which are introduced in this section will be used to explain how the prototypes and the study were developed. It starts explaining the UEFI specification version 2.1, which is the one used on the GEP boards' BIOS. The different types of services, drivers, applications and protocols are explained. Then a special UEFI application, the UEFI Shell, is introduced. The most used implementation of the UEFI specification is the EFI Development Kit (EDK). It is the environment which is going to be used to develop the prototypes. It is also introduced in this chapter. Finally, some practical information on how to install, build and use the EDK and how to build, test and execute UEFI shell applications is provided.

2.1 UEFI specification version 2.1

As explained in section 1.7.1, UEFI version 2.1 was released on January 2007. This specification has evolved over the years and the last version is 2.3.1, released in 2011. Nevertheless, GEP boards support UEFI specification version 2.1 [61]. Therefore, the next sections aim to explain how this specification is designed and how it can be used. The specification and its implementations are written using the *C programming language*.

2.1.1 Introduction

The UEFI specification describes an interface between the OS and a platform's firmware. It is depicted on Figure 2.1. It consists of data tables which contain platform-related information, boot service calls and runtime service calls which the OS and the OS loader can use. This combination provides a standard environment for booting an OS. The UEFI specification is an abstract specification of the

software-visible interfaces which the platform and firmware present to the OS. There are three main benefits derived from using this approach:

- It increases OS portability and flexibility. An OS intended to run on a platform compatible with the specification can boot on a variety of system designs without needing to perform further modifications on the platform or the OS.
- It eases the introduction of new features and functionalities to improve platforms' capabilities. The new code will not need to be written on the OS boot sequence.
- It allows upgrading legacy devices and firmware code over time. If the new devices and code comply with the same abstract interface there will be no impact on the OS boot support code.

A wide range of hardware platforms (from mobile systems to servers) can support this specification. It provides a core set of basic services and a selection of protocol interfaces. The protocol interfaces evolve over time and give support to many different tasks, from memory management to network features' support.

2.1.2 Design overview

The UEFI specification [61] defines an *API* which provides interfaces to UEFI services, which are implemented as C functions. It provides access to two types of services:

Boot services They can only be used during boot time. System resources are owned by the firmware during the boot stage. They are controlled through the boot services' interface functions. UEFI applications use boot services' functions to allocate memory and access devices. They are used until the OS loader loads enough of its own environment and takes control of the system. Their function is to assist the OS loader when it boots the OS. Boot services are terminated with a call to *ExitBootServices()* (which is one of the boot services) made by the OS loader.

Runtime services Their goal is to abstract minor parts of the hardware implementation of the platform from the OS. They are available during the pre-boot stage and after the OS is loaded. The memory used by this type of services is runtime memory (See Figure 2.1). This memory must be reserved and will not be used by the OS. This memory is always available to a UEFI function and it is never directly manipulated by the OS or its components. UEFI is responsible for defining hardware resources used by these types of services.

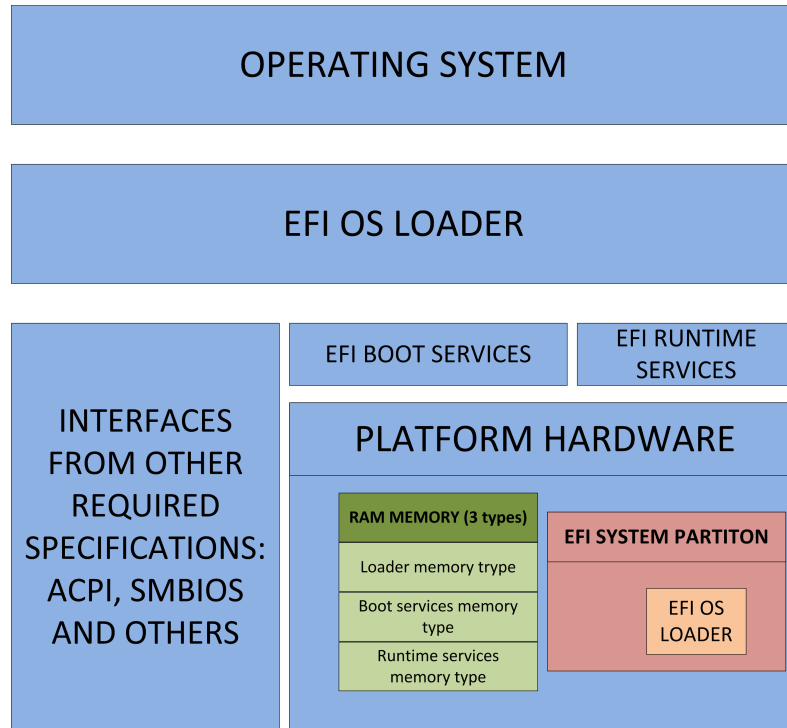


Figure 2.1: UEFI Overview.

Figure 2.1 shows the interactions between several components of a system supporting the UEFI specification. First, the platform’s firmware retrieves and executes the OS loader *binary image* from the System Partition. This partition is usually stored in secondary storage devices. Nevertheless, it can also be stored in ROM memory. Then, the OS loader finishes booting the OS. During this process, several platform components might need to be surveyed, comprehended or initialized. The boot loader can use the boot services and interfaces to accomplish those tasks. The figure also shows three types of UEFI RAM memory: loader memory, runtime services memory and boot services memory. They are logical types of memory, which have different purposes on UEFI systems.

Driver model

It is important to distinguish between an OS driver and a UEFI driver. A UEFI driver is a modular piece of code that runs in the pre-boot environment. They are used to provide software-derived, platform-specific services and to manage hardware buses or devices in the platform. Drivers can be accessed through UEFI protocols. UEFI drivers should not be used to replace OS-specific drivers.

The UEFI Driver model can be adapted to any type of bus or device. As it will be explained later, the RAM Disk drive prototype was designed as a UEFI driver.

Booting sequence

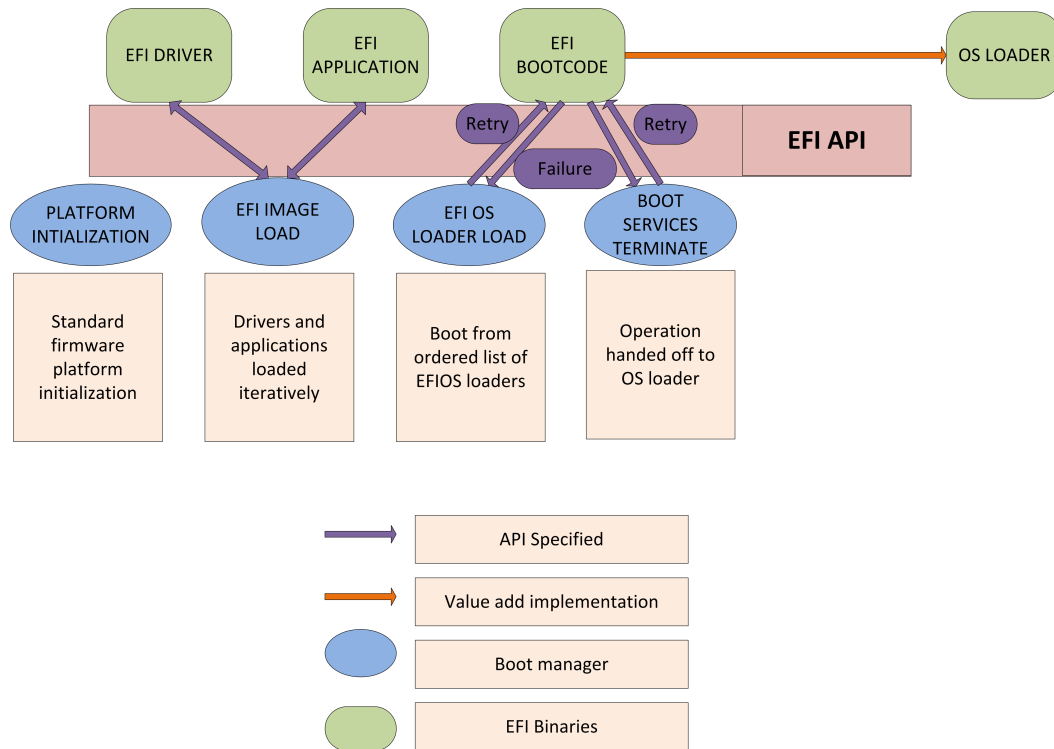


Figure 2.2: UEFI Booting sequence.

Figure 2.2 shows UEFI's booting sequence. It starts with the Standard firmware platform initialization which ends transferring execution control to the EFI Image Load. The EFI Image Load enters in a loop which iteratively loads EFI drivers (also called UEFI drivers) and applications. EFI applications (also called UEFI applications) are modular pieces of code like EFI drivers are. Nevertheless, the system treats them differently, as it will be explained later.

When all the drivers and applications have been loaded, the execution control is transferred to the EFI OS loader. The EFI OS loader has an ordered list of EFI OS loaders which it will use to load an OS. Instead of loading an OS it can also start a *UEFI Shell*. The UEFI Shell is explained in section 2.2 and it is the environment in which the Remote File Access System is going to be executed.

2.1.3 UEFI Images

UEFI Images are files which contain executable code and which are defined by UEFI. UEFI uses a subset of the *PE32+* image format (an explanation about this type of image format can be found on [37]). Nevertheless it has a modified header

signature. This modification allows UEFI images to be distinguished from the usual PE32 executables. There are three types of UEFI images:

- Applications
- Boot services' drivers
- Runtime services' drivers

The three types of images differentiate themselves from the others in the type of memory they are loaded into. The different types UEFI memory can be found in Figure 2.1:

- Applications are loaded in loader memory.
- Boot services' drivers are loaded in boot services memory.
- Runtime services' drivers are loaded in runtime memory.

These types of memory are logical constructions used by UEFI to establish certain policies concerning writing, reading and executing access permissions. In addition, when the *image's entry point* returns the system's execution control, the system behaves differently for drivers and applications. When control is returned from an application's entry point, the application is unloaded from memory. On the other hand, drivers are only unloaded when they terminate their execution with a UEFI error code. Images are designed and built to be run on three different types of architectures.

IA-32 Which stands for *Intel Architecture, 32-bit*. It is Intel's most commercially successful *instruction-set* architecture which consists on an extension of the *x86 architecture* [26].

IA-64 Also called *Itanium Architecture*. It is a 64-bit architecture.

x64 Also called *x86-x64*. It is an extension of the x86 instruction-set which provides 64-bit general purpose registers among other enhancements. The Intel implementation of this specification is called Intel 64 [26].

The prototypes have only been tested on x64 GEP boards. All the images are loaded into memory using the *LoadImage()* UEFI boot service.

Applications

They are loaded by other UEFI applications or by the *boot manager*. The boot manager is a piece of firmware which allows loading UEFI applications or UEFI drivers. Applications might be loaded with additional options. The load options are stored in nonvolatile memory.

An special type of applications are the OS loaders. If they are successfully loaded, they can take control of the system by using the boot service *ExitBootServices()*. After that call, all boot services are terminated and the OS loader is responsible for the continued operation of the system. The Remote File Access System will run on the pre-boot environment, this is, before calling *ExitBootServices()*. Nevertheless, the RAM disk drive can persist on the memory after the *ExitBootServices()* call. Therefore, some mechanisms are designed to provide the OS the needed information to access it.

Drivers

They can also be loaded by the boot manager and other UEFI applications. As opposed to UEFI applications, they are only unloaded from memory if they return an error status code. UEFI status codes are a type of data defined by UEFI used to signal how an application or driver has finished its execution process. A complete description of UEFI status codes can be found in UEFI specification 2.1 Appendix D [61].

As it was stated in section 2.1.3 there are two types of drivers. Boot service drivers are only available before calling the *ExitBootServices()* boot service. When the call is made all the memory resources which were allocated to these types of drivers are released. Runtime drivers are available after calling *ExitBootServices()*.

2.1.4 Protocols

Protocols are UEFI services which are usually implemented using UEFI drivers. Every protocol includes the following components:

- Its globally unique identifier (*GUID*).
- The Protocol Interface structure.
- The Protocol Services.

Figure 2.3 explains the construction of a protocol:

- Drivers support functions specific to one or more protocol implementations. They must use the *InstallProtocolInterface()* boot service call to register them in the system.
- Then the firmware returns the Protocol Interface for the selected protocol. It will be used to invoke the protocol specific services.
- The UEFI drivers keep device-specific private context with protocol interfaces.

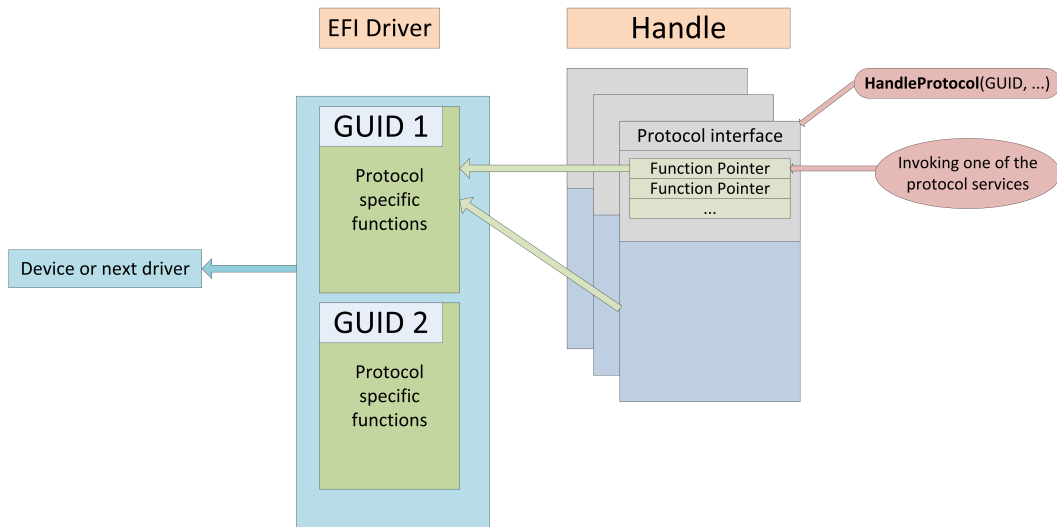


Figure 2.3: Construction of a protocol.

Protocols are used for a wide variety of tasks such as:

- Handling binary images.
- Handling file systems.
- Handling block devices (used in the RAM drive prototype, as it will be explained in the following chapters).
- Network functions (used in the TFTP client prototype, as it will be explained in the following chapters).
- Debugging.

2.1.5 How UEFI drivers are implemented on GEP BIOS

In order to write and build drivers which are going to be loaded and run on GEP boards several files are necessary:

Make file

This file is used to instruct the compiler how to compile the application to produce the driver's binary image which will be loaded in the BIOS. It contains instructions on how to build the driver, the libraries which should be included to build the driver and the driver's entry point function.

Simple Declarative File

The Simple Declarative File or *SDL file* contains several variables such as the driver's build path or the name of the binary file which will be created when the driver is built. SDL is a cross-platform language for defining basic data structures.

Dependency Expression file

The Dependency Expression file or *DXS file* contains the protocol dependencies (see section 2.1.4) needed by the RAM Disk driver.

Component Information File

GEP BIOS system is composed by several components. These components are drivers or applications which are loaded during the GEP platform's initialization. They are used to access and use UEFI protocols and services (GEP BIOS system supports UEFI 2.1 specification [61]). They are also used to perform memory and hardware tests during the platform's initialization. The Component Information File or *CIF file* is used to declare all the driver's source files. It is also used to declare the component's name, the component's type and its location in the BIOS system.

Source code and header files

They contain the actual source code of the driver and they are written using the C programming language.

2.2 The UEFI Shell

The UEFI Shell is a special UEFI application which is stored in the GEP boards' BIOS ROM. After the platform has been booted, but before an OS has been loaded, the UEFI Shell application can be invoked. The UEFI Shell is an optional application which is not used in all UEFI systems. It is used in UEFI systems which need to access a platform during their pre-boot stage, before an OS has been loaded. It provides a simple and interactive environment which allows the following, as shown by [23]:

- Loading drivers before an OS has been loaded.

- Launching applications before an OS has been loaded.
- Using a set of basic commands to manage files, system environment variables, and other elements of UEFI before an OS has been loaded.
- Booting OSs.

According to [62] it supports:

- An interactive command-line interface.
- Scripting.
- A standard set of commands.

Figure 2.4 shows its appearance.



```

EFI Shell version 1.10 (8192.1)
Current running mode 1.1.2
Device mapping table
  fs0 :UnknownDevice - Alias F4 blk0
        VenHu (58C518B1-76F3-11D4-BCA-0080C73C8881) /VenHu (0C95A92F-A006-11D4-BC
FA-0080C73C8881)
  fsnt0 :UnknownDevice - Alias F5
        VenHu (58C518B1-76F3-11D4-BCA-0080C73C8881) /VenHu (0C95A935-A006-11D4-BC
FA-0080C73C8881)
  blk0 :UnknownDevice - Alias F4 fs0
        VenHu (58C518B1-76F3-11D4-BCA-0080C73C8881) /VenHu (0C95A92F-A006-11D4-BC
FA-0080C73C8881)

Press ESC in 5 seconds to skip startup.nsh, any other key to continue.
Shell> _

```

Figure 2.4: UEFI Shell appearance

There are three types of commands [23]: internal, external and script-only commands. Internal commands can be issued entering the name of the command at the command prompt. External commands are executed using binary image files stored in a file system which could be implemented on an external drive or on an internal drive. Finally, script-only commands can only be executed when using script files.

As it was shown in section 2.1.3, applications can be loaded by the boot manager or by other applications. The TFTP client prototype is built as a UEFI Shell internal command, as it will be shown in the following chapters. The UEFI Shell is loaded by the boot manager.

2.3 EDK

2.3.1 Intel Platform Innovation Framework for UEFI

The *Intel Platform Innovation Framework for UEFI* (also called the Framework) is a firmware implementation conforming to the UEFI specification, as explained by [19]. It is a set of robust architectural interfaces, implemented in C. Intel recommends this implementation of the UEFI specification for platforms based on members of the Intel architecture family. In addition to implementing the UEFI specification, it performs a full range of operations needed to initialize a platform. The open-source part of the Framework is called the EDK project.

2.3.2 The EDK project

The EDK (EFI Developers Kit) is the open-source part of the Intel Platform Innovation Framework for EFI, as explained by [59]. It includes the *Foundation Code* of the Framework, sample drivers and project-specific build tools.

The Foundation Code consists on the core interoperability interfaces between different firmware modules and the Framework. It is released under a BSD license from Intel. It is also used as a development kit for developing, debugging and testing EFI drivers and applications used in the Framework environment, as shown by [58].

There are two EDK projects. The *EDK1 project* complies with UEFI Specification version 2.1. It includes an UEFI Shell implementation. The *EDK2 project* complies with UEFI Specification version 2.3. The EDK version used on GEP platforms is EDK1 and that will be the version referred by this document when using the term *EDK*.

2.4 Setting up the EDK environment

After performing the literature review it was necessary to set up an environment to write and build the EDK, the UEFI Shell and the UEFI applications. It was also necessary to learn how the environment worked using several tutorials. Some sample applications were also built and tested.

2.4.1 EDK Requirements

Before testing UEFI applications and drivers on the GEP platforms' 64-bit environment they were tested in an emulated environment to debug them. Although a 64-bit emulated environment would have been the best choice, EDK only provides a 32-bit emulated environment [60]. Nevertheless, applications and drivers can be built for 32-bit and 64-bit environments using the EDK. Consequently, the applications and drivers were first build for 32-bit environments to debug the main

problems. When they worked properly on the 32-bit emulated environment, they were built for 64-bit environments and then they were tested on the real GEP platforms.

This emulated environment has the following system requirements [25] (which are also EDK's system requirements when using the emulated environment):

Hardware requirements

- At least 256MB of RAM memory.
- At least 500MB of free space on a hard drive.

Software requirements

- Microsoft Windows XP or higher OS.
- Windows Visual Studio 2005 Professional (Ver 8.0) or Windows Visual Studio .NET 2003 Enterprise.
- Microsoft Windows Server* 2003 Driver Development Kit (DDK), build 3790.

2.4.2 Equipment

Hardware

It consists on a HP laptop which was used to write UEFI applications and drivers, build them and test them on a 32-bit emulated environment. It has the following features:

- Intel Core i5 CPU processor. 2.40 GHz.
- 2.00 GB RAM memory.
- 50GB free space on hard drive.
- 32-bit Operating System.

Software

The following software was used on the laptop:

- OS: Microsoft Windows Vista Enterprise Edition with Service Pack 2.

- Microsoft Visual Studio .NET* 2003 Enterprise (7.1) .
- Microsoft Windows Server* 2003 Driver Development Kit (DDK), build 3790.
- Eclipse IDE for C/C++ Developers. Version: Helios Service Release 2.

2.4.3 Installing the EDK

In order to set up the EDK environment and learn how to use it the EDK Getting Started Guide was followed [21]. The source code was downloaded. It is a software snapshot downloaded from Sourceforge website ¹. The downloaded version which was used on the GEP boards was the most recent: *version 20100527*.

2.4.4 Running a 32-bit emulator

One of the EDK features is a 32-bit emulator. It is called the *Nt32* emulator. It is used to create a virtual platform where UEFI applications and drivers (including the UEFI Shell) can be tested. It is necessary to configure the system using Microsoft Visual Studio .NET 2003 before using it. Then an environment variable called *EDK_SOURCE* must be set to the folder in which the EDK has been installed.

If access to a file system is needed it is necessary to configure a batch file named *System.cmd* which runs a Windows shell script to set more environment variables to manage emulated devices and configure the NT32 environment. The *System.cmd* file is only used to set emulation parameters, it is not used when building EDK for a real platform. It can be used to map a physical serial port into the emulator, map a Windows file system to the emulator or map Windows virtual and physical disks to the emulator. During the setup process it was only used to map a Windows file system to the emulator in order to be able to access the sample applications' binaries which were going to be tested.

Then a tool called *nmake*, included in Microsoft Visual Studio .NET 2003 was used to compile and run the emulation. The *nmake* tool needs a *makefile* to start building the EDK. This makefile is provided in the EDK source code and should not be changed. For configuring the EDK build process on a certain platform a build description file (also called *DSC file*) must be edited.

Building the EDK on other platforms

The EDK can also be built for Itanium and x64 platforms. The process which has to be followed is the same one as in the Nt32 emulation. The only difference is that the *System.cmd* file does not need to be edited because in this case the system is being built for a real platform, not a emulation. Therefore, in this case, the EDK

¹<http://sourceforge.net/projects/efidevkit/files/Releases/Development%20Snapshots/>

build will be used on real platforms. Each platform (Nt32, Itanium and x64) has its own DSC file although EDK's source code files are the same for all the platforms.

2.5 Writing, building and testing UEFI Shell applications

The UEFI Shell implementation started as EDK's sub-project as shown by [22]. Nevertheless, it evolved as an independent project.

2.5.1 UEFI Shell applications' components

An UEFI Shell application needs two files, (in addition to the C header and C source files) to be built:

String Resource .uni File

It is an optional file and it contains Unicode string resources. The file suffix is *.uni*. It is only needed if the application is going to use fixed string resources. It contains several key words with their associated strings which can be specified in several languages. The file is compiled to produce a C source file which contains all the strings formatted in a specific format in an array. It is useful because it allows to print the same messages in different languages. This can be used to build a language-independent application.

Component Description .inf File

It is used to instruct the compiler on how to compile the application to produce a binary image. It has several sections:

- **DEFINES Section:** Used to specify the application's name and its unique identifier (GUID).
- **SOURCES.COMMON Section:** Used to specify the source files that will be used to compile the application.
- **INCLUDES.COMMON Section:** Specifies the header files which are used by the application.
- **NMAKE.COMMON Section:** Specifies the application's entry point function, which is the first function which will be executed. It is also used to instruct the compiler to use the component description file's string resources.

Figure 2.5 shows the relation between the different UEFI shell applications' files.

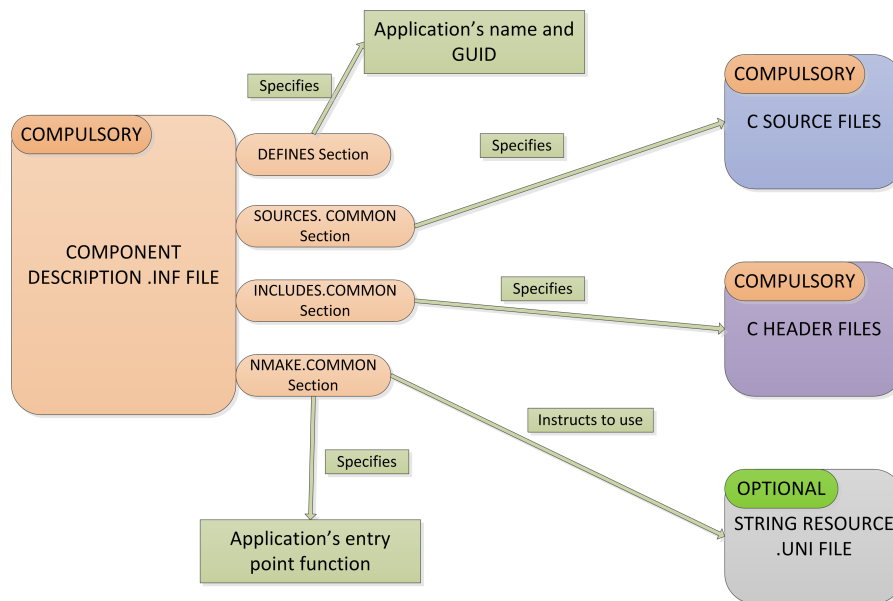


Figure 2.5: UEFI shell applications' files

2.5.2 How to build external UEFI applications

Once all the files have been written, several steps must be followed to build an external application for a x64 platform, as shown by [22]:

- The x64 *DSC file* must be edited to build the shell's library.
- The application which is going to be built must also be stated in the *DSC file*.
- The `nmake` tool must be used to build the EDK project. The UEFI shell is also built using the EDK environment. Nevertheless, its binary file is stored in the GEP BIOS ROM.

The application's binary image is also built using the EDK environment. When the application is built as an external command, its binary file is created separately. This binary file is stored on a storage media and then it is loaded using the UEFI shell. When it is built as an internal command, its binary file is embedded in the shell's binary file.

How to turn external commands into internal commands

External commands are useful during the software developing phase, when they are being tested and adjusted until they work properly. Nevertheless, an external drive is needed to make them available for the platform. Therefore, once an application

is completed, it should be included as a shell internal command. Building the BIOS with the shell, and flashing it into the ROM memory takes much more time than simply building an external application. Several steps must be followed to turn an external command into an internal command:

- The external application must be removed from the *DSC file*. This step is not stated in [22], but if it is not followed, the shell will not be built due to software inconsistencies.
- The Shell has also a component description file. Its SOURCES.COMMON section must be edited to include the applications' source files. To include the application's header files the INCLUDE.COMMON section must be edited.
- One of the shell's components is the command dispatcher. It loads external applications and internal commands. Therefore, the external application's entry point must be declared in the dispatcher's source code.

How to build and run some sample applications

The source of several UEFI Shell sample applications is provided in [22]. These applications were built, executed and modified to learn how to use the EDK environment. They are the following:

HelloWorld application A simple application which prints a message on the screen. Used to learn the basic steps needed to write an application.

UseArg application Used to learn how to access the command line arguments in an application.

UseArg application Used to learn how to set and read environment variables.

Chapter 3

File transfer client application prototype

This chapter describes the file transfer client application prototype. It starts stating the prototype's requirements and the state of the art: available solutions and services. A first FTP prototype is built adapting a FTP existing solution. The final TFTP prototype is also described. The chapter explains the prototype's design choices and implementation's difficulties. It also describes how it was tested and evaluated. Finally, some conclusions about the prototype and possible future work are commented.

3.1 Requirements and state of the art

3.1.1 Requirements

The aim of this prototype is implementing the file transfer application of the Remote File Access System explained in section 1.2. These are its requirements:

- The application must be set up during the GEP platform's initialization and will be run during pre-boot time.
- The application must work without external peripherals.
- It must be a UEFI Shell application.
- It will be used in Ericsson's laboratory environment, therefore security is not a concern.
- It must have command line support and it will be able to be executed by human beings or by other software applications.
- Although the application is a client, a proper server application must be selected and set up on one of the laboratory servers.

Due to the time limitations of the project it was critical to discover if there was any available open source solution and which were the available services to serve as the foundation for a new application.

3.1.2 Existing solutions

In order to find out if a file transfer application open source project had already been developed the following sources of information were analyzed:

- Tianocore's webpage: the website of the community which supports and develops the open source components of Intel's implementation of UEFI ¹.
- EDK development email lists ².
- Previous Intel EFI projects.
- EDK's source code.

There was only one previous open source implementation of a file transfer application for UEFI-based systems. It was found in a Tianocore unmaintained project called *EFI Toolkit*.

EFI Toolkit

According to [57], the EFI Toolkit consists on several tools used to allow rapid porting and development of UEFI applications, protocols and drivers.

Among all its functionalities it offers:

- A reduced version of the *C Standard library*.
- A TCP/IP_{v4} network stack implementation (composed by a UEFI protocol and a socket library).
- Several network utilities: A FTP client, a *DHCP* client (built as a UEFI driver), an *ifconfig* application, a *route* application and a *ping* application.

The description of the features and the limitations of the network stack and the network utilities, as well as an adaptation which was needed to use the EFI's FTP client in GEP systems are provided in section 3.2.

¹<http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=Welcome>

²<http://feishare.com/efimail/index.html>

3.1.3 Available services to build a file transfer client application

The UEFI Specification version 2.3 [63] defines an EFI FTP driver which could be used to implement a FTP client. It defines functions to connect and close connections with servers, configure FTP parameters, read and write files and manage remote directories. Nevertheless, this protocol is not implemented on GEP boards. The EDK2 source code (which supports the UEFI Specification version 2.3) has not implemented this protocol yet. The EDK2 project leaders were contacted by email in order to get some feedback about this issue. According to the information they provided, the FTP protocol is not going to be implemented in the short term (several months). They were not sure about the possibility of implementing it in the long term. They stated that the *Multicast* TFTP driver implementation was enough to cover the current needed file transfer functionalities.

Among the protocols which are available on the GEP platforms (implemented as UEFI drivers) there is a TCP/IP version 4 network stack which includes:

- An *ARP* protocol.
- A DHCP protocol.
- A TCP and a UDP protocol.
- A Multicast TFTP protocol

The Multicast TFTP protocol provides several functions which allow getting information from a TFTP server, configuring and parsing TFTP options, reading and writing files.

3.2 First FTP prototype using the EFI Toolkit

The TCP/IP network stack implemented in the EFI Toolkit uses a lighter version of the Standard C library to be built, as [20] states. It is built using the EFI Toolkit tools and it generates a binary image file which can be loaded as a UEFI driver.

It is a standalone network stack which does not use any of the functionalities provided by the GEP boards. In order to use this driver it should be included in GEP boards' BIOS ROM as a binary. Consequently, two separated network stacks would be stored in the ROM. Both would perform the same functions. This is a great disadvantage due to the memory size limitations present in GEP boards. The GEP BIOS TCP/IP stack based on UEFI protocols is enough to implement any network application.

The only advantage of using the Toolkit's TCP/IP stack is that it allows programming using standard C sockets. The TCP/IP stack which is part of the GEP BIOS offers its services through UEFI protocols which are described in [61]. These protocols are a much lower level interface to the TCP/IP services. Therefore, it is more complex to write applications using them. In addition, being able to use standard C sockets would allow Ericsson to port a lot of open source network applications without modifying their source code. These applications are already implemented and working on several OSs, the Toolkit's FTP application is one of them. Nevertheless they cannot work on pre-boot environments unless a TCP/IP stack which supports standard C sockets is implemented in the platforms.

There is another driver in the EFI Toolkit which launches a DHCP client application when it is loaded. Network configuration in Ericsson's lab is done using DHCP servers so this functionality is really valuable. Nevertheless it only works on the first Ethernet interface of the platform in which it is loaded (in this case, in the first Ethernet interface of the GEP board). GEP boards have more than one Ethernet interface. As shown in Appendix C, a GEP board has 5 1-Gigabit Ethernet interfaces (although one of them is only used for debugging).

In addition, this driver needs other configuration files which should be placed in a default folder in the working directory. This implies that each time the working directory is changed, that files should be copied to the new location in the file system. Consequently, it is harder to automate its configuration.

There is an *ifconfig* application which could be used to configure a link layer interface manually. There is also a *route* application to setup a gateway for the board's network interfaces and a *ping* utility to check the connectivity of the platform with the network. All these tools worked properly.

The EFI Toolkit's FTP client application can be found in the Berkeley Software Distribution (*BSD*) version 8.6. It is a complete client which supports nearly all the existing FTP options and works with every FTP server. This application, as well as the other utilities and drivers, has been built using a reduced version of the C Standard library. This is a great disadvantage because the built binaries need too much memory to be stored in the BIOS' ROM.

3.2.1 Adaptation to the GEP environment

Solving the DHCP driver problem using scripts

To tackle the problem concerning the inability of the DHCP client to work on every link layer interface, an UEFI shell batch program was created. These programs, also called shell scripts, are written in ASCII text files with a *.nsh* file name extension. They are used to execute several commands sequentially, as shown by [62] and

[23]. According to RFC 2132 [12], when a client tries to get DHCP configuration information from a server it must send a DHCP Discover message to request the DHCP configuration data from the active DHCP servers, with 0.0.0.0 as the source network IP address and 255.255.255.255 as the destination network IP address. The DHCP driver's source code invokes a shell script during its loading process.

This script invokes the Toolkit's *ifconfig* utility to configure the link layer interface with the lowest Ethernet identifier with a 0.0.0.0 network IP address. A new batch script was created and the driver's script was modified to be able to select the interface which needed to be configured.

The new batch script requires an argument to be invoked: the link layer interface Ethernet identifier which is going to be configured using DHCP. Then it stores it in a new shell variable that it creates. The first DHCP script was modified to read the shell variable in order to select the interface which was going to be configured using *ifconfig*.

Solving the TCP/IP driver problem

It was impossible to load the Toolkit's TCP/IP driver on the GEP system. Several modifications were performed to the driver's source code to solve the error which appeared during the loading process. The problem was caused by a memory limitation. Due to this limitation, variables whose size was bigger than 4098 bytes (a hard-coded value) could not be loaded, as shown by [3]. It was solved increasing the hard-coded value to 100000 bytes. Therefore it was finally possible to load the TCP/IP driver in the GEP boards.

Nevertheless when this system started to be adapted and tested there were only GEP boards with an older BIOS firmware version available for testing the system. The client was tested on one of these boards to connect to a BSD FTP server installed on one of the laboratory servers. It was able to connect, scan directories, download files and upload files but it still needed an external drive to work.

Lately some GEP boards with the new BIOS firmware version were provided to test the system. When trying to load the TCP/IP driver on these boards an error occurred and the system halted. After running some tests it was discovered that the current firmware supported some legacy options which were not compatible with the TCP/IP driver and the DHCP driver. It was impossible to change the GEP BIOS firmware.

In order to use the Toolkit's FTP client, a TCP/IP driver which supports standard C sockets is needed. One possibility to solve this would have been to build the socket-related functionalities in top of the already available UEFI TCP/IP driver installed on the GEB BIOS. Nevertheless, as it is shown in section 3.2.2, there were

other problems with this prototype.

3.2.2 Problems with the first prototype

The Toolkit's FTP application needs a TCP/IP driver which supports sockets to be used. As it was shown in the previous section, it was impossible to use the one provided by the Toolkit in the boards with the newest firmware. In addition there were other problems:

- The system required an external drive. One of the Remote File Access System requirements is that it must work on GEP boards without external peripherals. This problem might have been solved by modifying the DHCP driver's source code, in order to avoid using external configuration files, by hard-coding the needed values in the source code. In addition all the external tools must have been turned into internal commands, which is not a trivial task because they are not built as the other shell applications and they need to be compiled using the EFI Toolkit environment.
- The system required too much ROM memory. The utilities which needed to be ported to internal commands would have been stored in the BIOS shell which is stored in the GEP BIOS ROM. The drivers would have been stored also in this memory space. The total needed space including all the utilities and the drivers was 3.41 megabytes and there was only one megabyte available. The BIOS ROM had not enough memory available to store the utilities and it was not possible to add more ROM memory. The cause of this problem is that all the utilities and drivers are built using a reduced version of the C Standard library. This reduced version still requires a lot of memory. A possible solution would have been to limit the functionalities of the FTP client and the utilities and build a reduced version of the libraries.

Due to all these problems it was decided to use the BIOS available services to design a different solution which could work independently of the BIOS firmware version of the GEP boards and without requiring an external drive.

3.3 Final TFTP Prototype Design

3.3.1 Selecting the file transfer protocol to be implemented

The file transfer protocol's choice determined the features, limitations and possible problems which might arise during the prototype's design and implementation. The following parameters determined the choice:

- Security was not a concern: Although it would add value to the prototype, security could only be used as a tie breaker parameter.
- Throughput: Throughput is not a critical parameter. The average file size which is going to be downloaded or uploaded is around tens of megabytes, according to the application's future users. It is acceptable to need tens of seconds to download them.
- Memory requirements: The prototype was going to be stored in the BIOS ROM, which is really limited. Therefore the desired prototype should be as light as possible. One megabyte is the memory limit.
- Time limitations: The prototype must be implemented according to the master thesis project schedule and time requirements.
- Reusing code: The prototype should use available functions and libraries instead of developing new ones when possible. In addition to be a good programming practice, it would help to comply with the memory limitations.

Secure file transfer protocols are much more complex than non secure ones. Consequently they need more memory. In addition, developing a prototype using one of these protocols might be unfeasible given the time limit constraint. Therefore, secure protocols were not eligible. The two possible options were FTP and TFTP.

FTP and TFTP comparison

FTP and TFTP are compared according to their RFCs [48] [52]:

- Security: Both are insecure protocols, neither of them uses encryption. FTP uses an authentication mechanism but the authentication information traverses the network in the clear, so it is not secure.
- Throughput: FTP is built on top of TCP. TCP provides a data stream service and it is a reliable protocol which uses sliding windows and acknowledgment messages. Therefore FTP does not need to acknowledge the data packets which are received.

TFTP works on top of UDP, which offers a non reliable datagram service. Consequently TFTP uses its own acknowledgments. It uses a stop-and-wait protocol to send blocks of data, which is less complex than a sliding windows protocol, but which also results on lower throughputs. In addition, if no additional options are used, it has a maximum data block size of 512 bytes. Therefore, after sending 512 bytes the server will stop and wait for the client's acknowledgment before continuing sending data blocks. Nevertheless this problem can be solved using the TFTP block size option extension.

This problem is specially critical when the network has a high round-trip delay time, which is the time it takes for a signal to be sent plus the time required to receive an acknowledgment for that signal. When bigger TFTP block sizes are used it is more probable that the packet which arrives to the client has been corrupted by the link noise and interferences. Nevertheless, the links used in Ericsson laboratory are secure channels with little noise and interferences and the distances to be covered are really short.

As a conclusion, FTP will always offer better throughputs than TFTP due to the use of TCP's sliding windows protocol to acknowledge the data. Nevertheless, when using higher TFTP block sizes in a channel with little noise and interferences, TFTP's throughput can be increased to achieve the required throughput.

- High memory requirements: Due to its simplicity, TFTP can be implemented using a very small amount of memory. In addition, there is a multicast TFTP driver available on GEP boards' BIOS. The specification of this driver can be found on the UEFI Specification version 2.1 [61] in section 25.2. The multicast functionality is optional and it is not needed for this prototype. Nonetheless, if a multicast TFTP client is needed in the future, it would be easier to upgrade an existing unicast prototype using this driver. FTP would require much more memory: it is more complex and there is no FTP driver available on BIOS boards.
- Time limitations: TFTP is less complex than FTP. On the other hand, as it was stated on the previous point, there is a Multicast TFTP driver available, therefore it would require less time to be implemented.
- Reusing code: Both protocols could reuse the TCP/IP network drivers and protocols available in GEP boards' BIOS firmware but only TFTP could take advantage of the Multicast TFTP driver and protocol.

The only reason to choose FTP over TFTP according to the chosen parameters would be its higher throughput. But as it was said at the beginning of this section, that is not an important concern as far as TFTP is able to offer an acceptable throughput.

In general, FTP offers more functionalities than TFTP in exchange for higher memory and complexity requirements. These are other differences, which are not critical for the prototype, but which are worth to be mentioned:

- FTP can be used interactively because it is a session-oriented protocol, while TFTP is not session-oriented and is more used on automatic operations without direct human intervention.

- FTP needs two data connections per file transfer (one for the data and another for the session control) while TFTP only needs one.
- FTP allows to send files, receive them, rename them, delete them and many other operations. TFTP only allows to send and receive files.
- FTP allows to browse directories remotely while TFTP does not.

FTP is superior to TFTP in many aspects. Nevertheless, due to the problem constraints and the project time limitations, TFTP offers a better solution to the problem stated in section 1.1.

3.3.2 The TFTP protocol

RFC 1350 [52] defines TFTP as a very simple protocol used to transfer files using UDP as its transport protocol. The RFC does not discard the possibility of using it on top of other datagram protocols. It was designed to be small and easy to implement. Its only functions are to write and read files to/from a remote server. As it was stated in the previous section, it cannot list directories and it does not provide user authentication.

As stated in section 1.7.2 it has three transfer modes, but this description will be focused on the octet mode which is the one which is used in the prototype. This mode is used to transfer raw 8 bit bytes.

Overview

One TFTP connection is initiated when a client requests to read or write a file. If the server grants the request, the connection is opened and the file will be sent in fixed blocks of 512 bytes (an extension option to overcome this problem will be explained later). Each packet must be acknowledged before the next packet is sent. Therefore, the sender only has to store one packet for retransmission, which is a really low memory requirement. In this protocol both machines are senders and receivers (the first one sends data and receives acknowledgments, and the second one sends acknowledgments and receives data).

The protocol uses timeouts to detect network failures. A packet with less than 512 bytes signals the end of the connection. Most errors cause the termination of the connection. Errors are treated with error packets which are not acknowledged or retransmitted, therefore timeouts are needed to detect lost error packets. There are three causes of errors:

- The server is not able to satisfy a request. For example the requested file might have not been found in the server.

- A packet that should not have been received. For example incorrectly formed packets.
- The address to a necessary resource is lost during the connection.

TFTP Header

The TFTP header consists of a 2 byte opcode field which indicates the packet's type. There are five types of packets according to the opcodes:

Opcode 1 Read request or RRQ

Opcode 2 Write request or WRQ

Opcode 3 Data DATA

Opcode 4 Acknowledgment or ACK

Opcode 5 Error

Both request packets have the structure shown in Figure 3.1. The filename contains the path to the requested file on the other machine and the mode can be *netascii*, *octet* or *mail*.

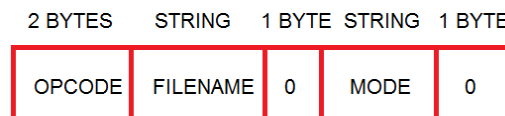


Figure 3.1: RRQ/WRQ Packet

The data packet is shown in Figure 3.2. The block numbers begin with one and increase by one for each new block of data which arrives. Therefore only one packet is needed to discriminate between new packets and duplicates.

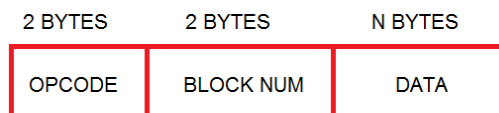


Figure 3.2: DATA Packet

TFTP packets are acknowledged unless a timeout occurs. Write request and data packets are acknowledged by ACK or ERROR packets. Read request and acknowledgments packets are acknowledged by data or error packets. Figure 3.3 shows the structure of an ACK packet. The block number in an acknowledgment packet echoes the block number of the data packet which is being acknowledged. The only



Figure 3.3: ACK Packet

case when an acknowledgment packet with a block number which equals zero is used is when it is used to acknowledge a write request packet.

Figure 3.4 shows the structure of error packets. The error code is an integer which indicates the nature of the error. The error message is an ASCII string intended to be read by humans.



Figure 3.4: ERROR Packet

Initial connection protocol and operation

A transfer is established by sending a read or write request and getting an acknowledgment packet for a write request or the first data packet for a read request. If the request has been denied an error packet is sent.

The server and the client must randomly choose a Transfer ID (TID) for themselves, which will be used during the connection. Each packet will have a source and destination TID associated with it. The TIDs are used to give TFTP the possibility to be supported by different datagram protocols. In the case of UDP, the TIDs will be used as the UDP source and destination ports.

A client randomly selects its TID and uses it as its source TID. Then it sends a request to the well known TID number 69. The server will take the request, generate a random TID which it will use as its source TID for the rest of the session. It will use the TID randomly generated by the client as the destination TID. Both TIDs will be used until the connection ends.

Normal termination

The end of a transfer is signaled by a DATA packet with a length between 0 and 511 bytes, which will be acknowledged with an acknowledgment packet. The sender of the acknowledgment can terminate the connection after sending the acknowledgment packet. The sender of the last DATA packet should wait until it receives the acknowledgment or until it times out.

Premature termination

When a request cannot be granted or some error occurs during the connection an error packet will be sent. It will not be acknowledged. Timeouts are also used to detect possible errors.

3.3.3 TFTP Option Extensions

RFC 2347 describes a method to enhance TFTP with an option negotiation which can occur before the file transfer is started [34]. It is a backward compatible extension to the TFTP protocol. It is a general purpose mechanism which can be used to negotiate several options.

To perform the backward compatible extension, the options are appended to the read request or write request packets as it can be seen in Figure 3.5.

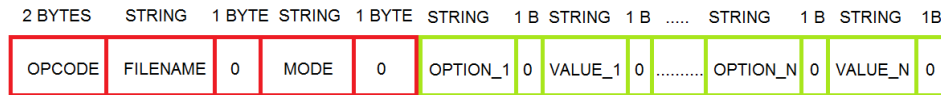


Figure 3.5: Request packet with options

Each option needs two fields, the option name and the option value. They are both ASCII strings. There can be several options which are negotiated in a request. All the fields of all the options are separated by 1 byte with all its bits set to 0. The order of the options is unimportant and the limit on the number of options is set by the TFTP packet size limit (512 bytes).

The RFC also defines a new type of TFTP packet, the Option Acknowledgment (OACK) packet [34]. It is used to accept an option request. The structure of an OACK packet can be seen on Figure 3.6.

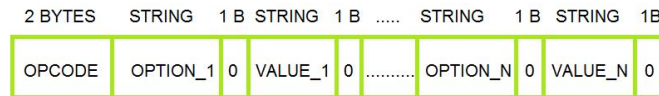


Figure 3.6: OACK Packet

OACK's opcode value is six. The option names must be the same as the requested ones. The option values may be changed according to the option specification.

Negotiation protocol

The client appends options at the end of its requests. An option may only be specified once. If the server supports the option negotiation it might respond with an OACK. The server includes each option it recognizes and accepts in the OACK. The option values that the client requested may be changed according to the option

specification. The server must not include options not requested by the client in the OACK.

The client must acknowledge the OACK by sending a normal acknowledgment packet with the block number set to 0 if it had initiated the connection using a read request. If it had initiated it sending a write request it should acknowledge the OACK sending the first DATA packet. If the client rejects the OACK it must send an error packet to terminate the transfer. The TFTP client prototype will use three types of options.

Timeout Interval Option

This option is defined in RFC 2349 [35]. It is used to negotiate the timeout interval between the client and the server. The option name field of the TFTP modified header must be set to *timeout* and the option value is the number of seconds. Valid values range between 1 and 255 seconds. If the server is willing to accept this option it will send an OACK to the client. The timeout value must be the same as the one specified by the client. This option is used in the prototype to try to set the server's timeout value.

Transfer Size Option

It is also defined in RFC 2349 [35]. The option name field of the TFTP modified header must be set to *tsize*. The value of this option represents the size of the file which is going to be transferred. In the case of a read request it is used to ask the server for the file size. Therefore the client will be able to allocate the memory that it needs for the file. In this case, when the client sends the request, it must set the option value to 0. The server will return the file size in the OACK.

When the client uses it in a write request packet, it must put the size of the file which it is going to transmit to the server. If the file is too large for the server to handle, it may abort the transfer with an error packet. Consequently, file transfers which are not going to be completed are avoided. This option is used in the prototype when sending read and write requests.

Block size option

It is defined in RFC 2348 [33]. This option is used to overcome the limitation of the maximum TFTP packet size, which is 512 bytes. When using bigger sizes, more memory will be needed by the sender of the file to store the last packet in case it needs to be retransmitted. The receiver of the file will need the same amount of memory independently of the employed TFTP packet size because it will not store packets which might be retransmitted. It only stores the file's data. But it also allows sending bigger files. The block numbers used in the data packets are limited by the 2 bytes of this field to 65535. The maximum file size is given by the product

of the maximum block number and the maximum TFTP data block size. When using the standard 512-byte limit, the file size is limited to 33.55 megabytes. The RFC states that the option value must be set to a value between 8 and 65464 bytes. Therefore, the file size limit can be increased to 4.29 gigabytes. The option name which must be set in the TFTP header is *blksize*.

For example, in Ethernet-based *LANs*, the block size can be set to 1428 bytes (which is Ethernet's *Maximum Transmission Unit* less the TFTP, UDP and IP header lengths) to avoid IP fragmentation and achieve better throughputs. The RFC ends with a study which compares the throughputs obtained using higher block sizes than 512 bytes. It demonstrates that the throughput is increased due to a reduction in the number of packets sent. Not only the number of data packets is reduced, the number of acknowledgment packets is also reduced by the same amount. If the block size exceeds the path MTU, IP fragmentation and reassembly will occur and more overhead will be added. Nevertheless this effect is only noticeable when traversing a great number of gateways.

3.3.4 Design choices - MTFTP4 protocol

The client was implemented as an UEFI Shell application. It treats the possible errors that can occur during its execution returning *EFI Status codes* [61] and printing error messages on the standard output. It is based on an UEFI protocol described in the UEFI specification [61] named the *MTFTP4* protocol. It is a Multicast TFTP over IP version 4 UEFI protocol. This protocol is already installed in the GEP BIOS' ROM and all its functions can be used by any shell application. This protocol provides basic services for the client side of unicast and/or multicast TFTP operations. It uses the UDP and IP drivers, which are also installed on the GEP BIOS' ROM, to perform its operations. The MTFTP4 functions which are used are the following:

The `Configure()` function

It is used to configure the MTFTP4 UEFI protocol instance. It provides permanent configuration data. The following information is provided when the protocol is configured:

- A flag to specify if the default network configuration (IP address, subnet mask and route table associated to a certain link layer interface) should be used on the client side. It is set to true in the prototype. Therefore the user does not need to provide that information, because it is automatically set.
- The local port which will be used by the client. In the prototype it is set to 0. When the port is set to 0, the platform will automatically look for an available UDP port for the TFTP connection.

- The IP address of the server. In the prototype it is extracted from the command line arguments provided by the user.
- The port in which the server will be listening to incoming connections. If the user does not provide this value, it will be set to 69, which is TFTP's default listening port.
- The number of times which the client will try to resend a TFTP packet after a timeout event occurs. If the user does not provide it, it will be set to 3. As it can be seen in the analysis section, this value assures that nearly any transaction is not aborted due to time out events.
- The client time out value, which is the number of seconds the client will wait for response to a packet before a timeout event occurs. If the user does not provide it, it will be set to 2 seconds. As it can be seen in the analysis section, this value assures that nearly any transaction is not aborted due to timeout events. When a timeout event occurs the try count is increased. If the packet has been sent the number of times specified by the try count value and another timeout event occurs the connection is terminated.

The Token data structure

There are other parameters which must be provided to perform a download or upload operation. These parameters are provided using a data structure named *Token*. They are provided each time a download or upload operation is started. They are the following:

- The name of the file which will be downloaded or uploaded (provided by the user with a command line parameter).
- The TFTP mode, which is always set to octet mode. It was chosen because it can be used with any type of file.
- A pointer to a function to check the received TFTP packets. In the prototype, this pointer points to a function which scans the packets searching for information about TFTP errors. Therefore, the user can be notified about the details of an error when it occurs.
- A pointer to the memory data buffer which is going to store the data of the file which is going to be downloaded or uploaded and the size of that buffer.
- The TFTP options used in the request. In the prototype, they depend on the actual type of transfer which is going to take place.

The ReadFile() function

It uses the permanent configuration data and the Token information to download a file.

The WriteFile() function

It uses the permanent configuration data and the Token information to upload a file to a TFTP server.

The GetInfo() function

This function sends a TFTP read request to the server with the chosen TFTP option extensions and then it returns the server's TFTP packet to the user. Finally it sends a TFTP error packet to the server to end the connection after the information has been collected. Its goal is to obtain information about the TFTP server's supported option extensions and other parameters. In the prototype, it is used with the transfer size option extension to discover the size of the files in the TFTP server. Therefore, the memory buffers which will be allocated to perform the transaction can be allocated with the appropriate quantity of memory.

The ParseOptions() function

It is used to analyze a TFTP packet which has been sent by the server to the client. In the prototype, it is used to analyze the packet which is received when the GetInfo() function is executed. Therefore, TFTP errors are detected and if there are no errors, the information about the server's option extensions are extracted from the OACK sent by the server. Therefore the file size on the server can be extracted.

3.4 Final TFTP Prototype implementation

3.4.1 Structure

The TFTP client application was designed and implemented as an internal UEFI shell command by the author of this master thesis. To be built, the required shell files explained in section 2.5.1 must be created. These files were designed and implemented by the author of this master thesis.

The String Resource file contains several strings to notify the user about errors that may occur. It also contains strings which provide help to the user: a line help string, which will be used when the user types the *help* command on the Shell and a verbose help string, which will be used when the user types the *help tftp* command on the Shell.

The **DEFINES** section of the Component Description file specifies the application's name for the UEFI Shell which is *TFTPClient*. Its **SOURCES.COMMON** section contains the source code files which will be explained in the following sections. Its **INCLUDES.COMMON** section contains the directories in which the header files of the application are stored. Finally, the **NMAKE.COMMON** section specifies the application's entry point which is *TFTPClientMain*.

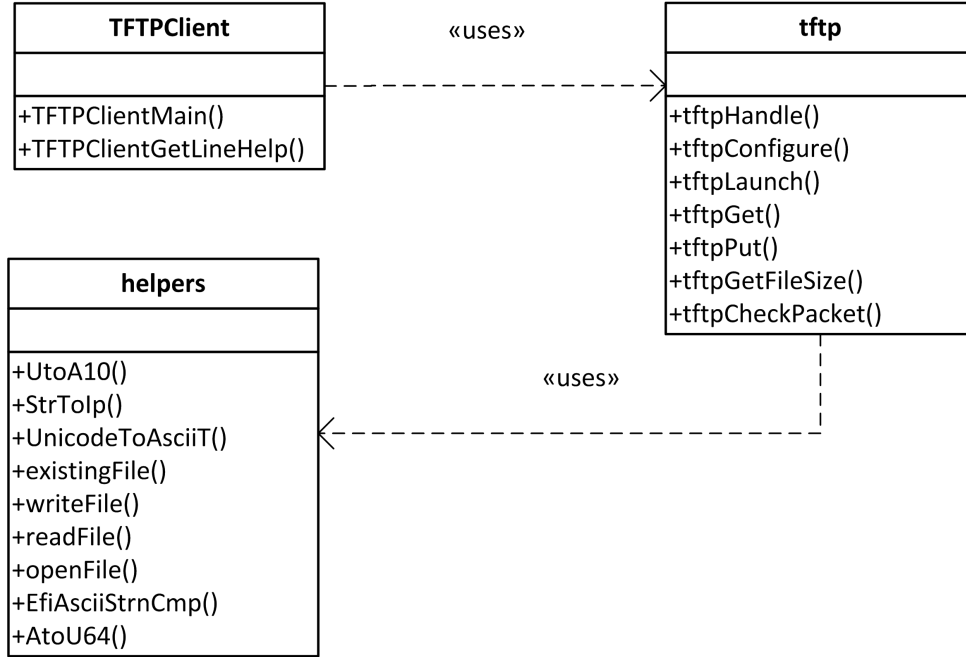


Figure 3.7: TFTP client structure

Figure 3.7 shows the division of the source code in three source code files. All these files and their functions and data structures were implemented by the author of this master thesis. Some of these functions use the MTFFTP4 protocol functions, which are already installed in the GEP BIOS' ROM, to perform their operations:

- **TFTPClient**: It provides the application's entry point *TFTPClientMain()* and the help line function *TFTPClientGetLineHelp()*.
- **tftp**: It is used to configure and use the MTFFTP4 protocol.
- **helpers**: It provides several helper functions which are used by the rest of the code.

Dividing the source code in these three files makes the code more legible and ordered. In addition, it is easier to reuse some of the functions.

Appendix D describes a summary of Figure 3.7 functions. It states who is their author (which is the author of this report). It also states which other support routines and functions are invoked by the functions depicted in Figure 3.7.

The TFTP prototype code will be allocated inside the GEP board's shell code. The GEP board's shell code is allocated in the GEP BIOS ROM. The UEFI MTFTP4 functions and the UEFI boot services used by the TFTP client have been implemented in the GEP BIOS code, stored in the GEP BIOS ROM, and they have been described in the UEFI Specification [61].

3.4.2 Overview

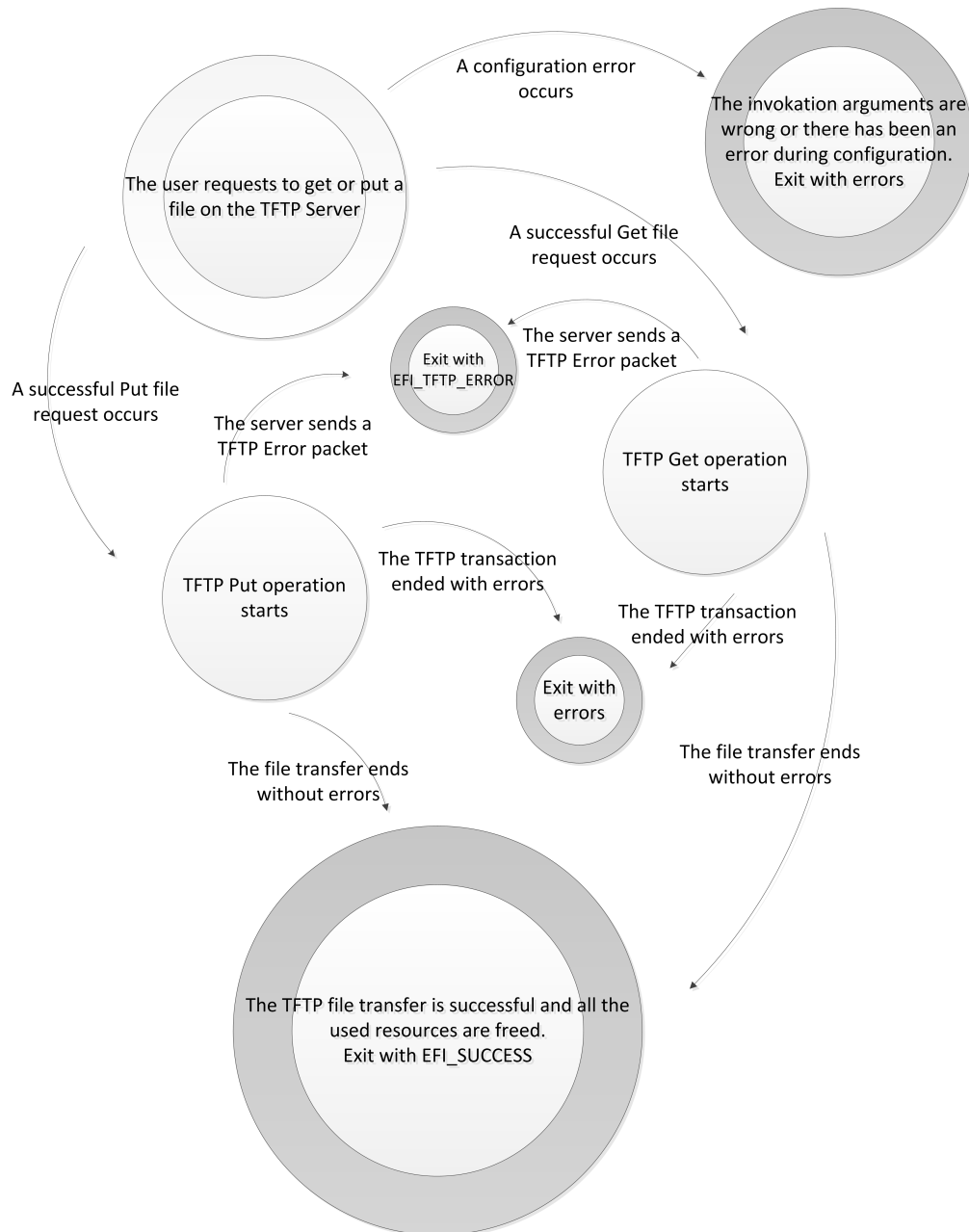


Figure 3.8: TFTP client overview

Figure 3.8 shows the prototype's overview. First, the user requests to get or put a file from a TFTP server. If the arguments used to invoke the application are incorrect or if there is any error during the configuration, the application will exit with an error status code. If the arguments are correct and the operation is successful, a get file operation or a put file operation will start.

Both operations can finish with three different results:

- With an *EFI_TFTP_ERROR* status code. It occurs when the server sends a TFTP error packet to the client. This could be due to several reasons: a requested file does not exist, the server has not space to store a file which is being uploaded by the client or others. In general it is related to an error on the server side.
- With an error status code different than *EFI_TFTP_ERROR*. It occurs when there is an error during the application's operation and it is not notified by the server using a TFTP error packet.
- With an *EFI_SUCCESS* status code. The file transfer was successful and the application ended without errors.

3.4.3 Initialization

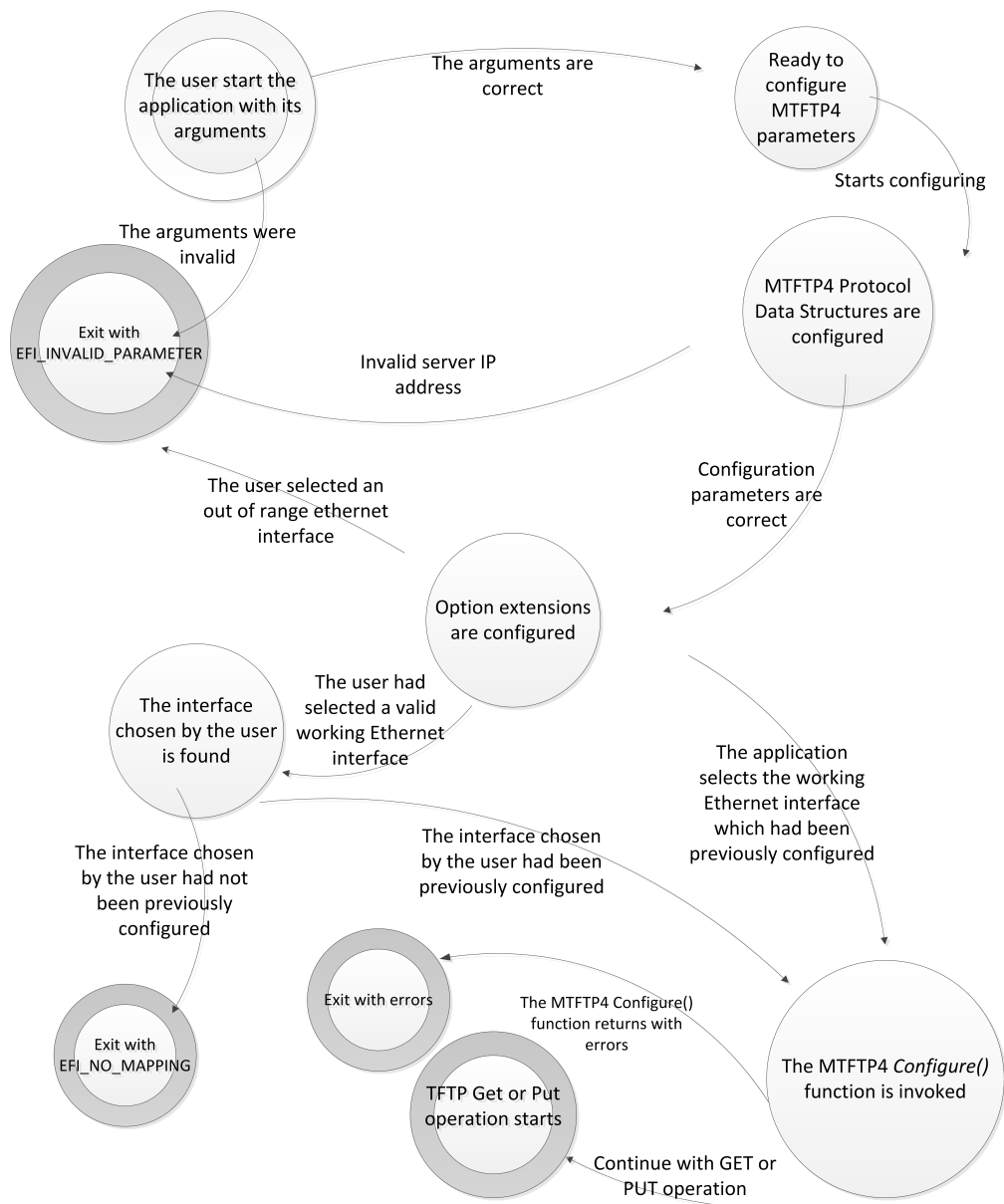


Figure 3.9: TFTP client initialization

Figure 3.9 shows the application initialization until it is ready to start a TFTP get or a TFTP put operation. The application starts when the user initiates it using the shell command *tftp*. Then the entry point (*TFTPClientMain*) is executed.

TFTPClientMain()

This function is the application's entry point and it is written in the *TFTPClient* source code file. The application is invoked with some mandatory parameters and with some optional parameters. The mandatory parameters are four:

- The type of TFTP operation: *get* or *put*.
- The remote path to the file which is going to be downloaded or uploaded.
- The local path to the file which is going to be downloaded or uploaded.
- The TFTP server's IP address.

The optional parameters are the following:

- The port in which the TFTP server will be listening. If it is not specified it is set to TFTP's well known port 69.
- The desired block size. Following RFC's 2348 [33] advice, it is set to the maximum by default to allow the biggest possible file size and improve the throughput. Section 3.3.3 contains more information about this option.
- The desired server's timeout value. It is used to set the TFTP's timeout interval option. The server might or might not use the requested value depending on its implementation. Consequently it is not used by default, the client's time out value is used instead.
- The requested Ethernet interface. By default, the client application will use the first available Ethernet interface which has been configured using the *ifconfig* command and which has the lowest identifier among the configured ones. The user might force the application to use a certain interface. That interface must have been previously configured using *ifconfig*.
- The client timeout value and client try count. They are set to the values specified in section 3.4.1 by default.

The optional parameters are specified with flags. For example to set the remote port to port 70 the following string should be added when executing the command: *-p 70*. The TFTP client application checks the validity of the arguments and will end with an *EFI_INVALID_PARAMETER* error status code in the following cases:

- An unknown flag has been used.

- There is a duplicated flag.
- The number of mandatory arguments is different than four.
- The requested command is not a *get* or *put* TFTP command.
- An invalid port has been specified. A valid port is a positive integer between 0 and 65535.
- An invalid block size has been specified. Valid block sizes are positive integers between 8 bytes and 65464 bytes, as explained by [33].
- The server timeout value or the client timeout value is invalid. Valid timeout values are between 0 and 255 seconds as explained by [35].
- The try count is not an unsigned integer.

All these parameters are checked to avoid possible problems during the application operation. If the arguments are correct the initialization continues invoking the *tftpHandle()* function.

tftpHandle() and tftpConfigure()

These functions are written in the *tftp* source code file. They are used to configure the required MTFTP4 protocol parameters before the get or put file operation starts. The functionality has been divided in two functions because the configuration process requires many lines of code. Dividing this functionality into two functions eases the application's maintenance and increases its legibility. The *tftpHandle()* function is invoked by the entry point.

It starts formatting the commands inputed by the user to the format required by the MTFTP4 protocol *Configure()* function and the MTFTP4 Token structure. For instance, all the text variables provided by the user are stored in a Unicode format (which requires 2 bytes for each character) and they have to be converted to the ASCII format (which only uses 1 byte per character). The *UnicodeToAsciiT()* helper function is used to accomplish that format change. This function was written by the author of this project.

Then it allocates memory for the data structures needed by the MTFTP4 protocol *Configure()* function and a MTFTP4 Token structure, which is created by the author of this report. The MTFTP4 Token structure is defined in the UEFI specification [61], as well as all the MTFTP4 functions. Nevertheless, the MTFTP4 *Configure()* function is not invoked by this function, it is invoked by the *tftpLaunch()* function. It transfers a pointer to the data structures to the *tftpConfigure()* function. This function will check if the server's IP address provided by the user is a correct IP address.

If it is not correct the application will exit with an *EFI_INVALID_PARAMETER* error status code. The function sets all the parameters required by the MTFTP4 protocol *Configure()* function with the values provided by the user and with default values when the user has not provided that data. It finishes completing some of the parameters needed by the Token structure with the values stated in section 3.3.4:

- The remote file name.
- The TFTP mode.
- The pointer to the function designed to check the received TFTP packets, which is the *tftpCheckPacket()* function.

Then the application returns to the *tftpHandle()* function. The function will complete the Token structure with the option extensions' data:

- The block size option extension is set to the default value unless the user has provided another value.
- The timeout option extension is only used if the user has requested it.
- The transfer size option extension is only set if the user has requested a TFTP put operation. It is used to notify the server about the size of the file which is going to be uploaded. Then, the server can decide if it wants to continue with the transmission of the file. The transfer size option extension is also used when the user requests a TFTP get operation, but in this case it will not be configured in the token, it will be configured when using the UEFI MTFTP4 protocol *GetInfo()* function.

Each Ethernet interface on the GEP platforms publishes a MTFTP4 handle to be able to access a MTFTP4 protocol instance. All the handles of all the interfaces are loaded using the UEFI *LocateHandleBuffer()* boot service. If no handles are found (something that may happen if there are not active Ethernet interfaces configured or the protocol is not present in the platform's BIOS) the application will return with an *EFI_NOT_FOUND* status code. The behavior of the function is different if the user has selected a certain interface to perform the file transfer or if the interface must be chosen by the application.

If the user selects the interface and it is a non-existing interface number the application will return with *EFI_INVALID_PARAMETER*. If the interface was valid, the UEFI *CreateChild()* boot service will be used to create a child handle for the MTFTP4 protocol instance. Finally, the UEFI *OpenProtocol()* boot service will be

used to get and initialize the MTFTP4 protocol instance associated to the requested interface.

The interface must have been configured previously with the *ifconfig* command. If it has not been configured the application will return with an *EFI_NO_MAPPING* status code.

The shell code, implemented in the GEP BIOS ROM, was also modified to automatically configure the first two Ethernet interfaces of the board using *ifconfig*. To do so, the source code file present in the shell's directory of the EDK *newshell/init.c* must be modified. The function *InitializeShell()*, which can be found in that source code file, must execute the required *ifconfig* commands using the *ShellExecute()* function described in [22].

Then, the *tftpLaunch()* function is invoked in order to perform the required TFTP operation. After the operation is finished the MTFTP4 protocol instance is closed and the child handle is destroyed. In case the user does not select the interface the behavior is slightly different. Instead of trying to configure a certain interface (specified by the user) the application will try to configure all the interfaces. Then it will call the *tftpLaunch()* function for each interface. If the *tftpLaunch()* function returns with an *EFI_NO_MAPPING* status code the next interface will be tested until an interface which had been previously configured is found. If there is not any configured interface the application will return with an *EFI_NO_MAPPING* status code. If it finds an interface which has been previously configured it exits the loop and goes to the end of the function. If several configured interfaces are found, the one with the lowest Ethernet identifier will be chosen.

In both cases (when the user requests a certain interface and when it does not do it) the *tftpHandle()* function ends its execution releasing the memory which had been allocated to handle the application parameters and the MTFTP4 data structures. Then, it returns its result status code to its caller. If the status code is *EFI_NOT_MAPPING* it prints a message on the standard output requesting the user to configure an interface using the *ifconfig* command. If there had been any errors during the MTFTP4 protocol configuration, the error status code which was originated during the configuration will be returned by the function. If there is no error during the configuration, the status code returned by the *tftpLaunch()* function to its caller (the *tftpHandle()* function) is returned.

tftpLaunch()

This function can be found on the *tftp* source code file, written by the author of this report. It is invoked by the *tftpHandle()* function. This separate function was designed for three purposes:

- Configuring the MTFTP4 protocol with the *Configure()* UEFI MTFTP4 protocol function and the data which was formatted and passed to this function by the *tftpHandle()* function. If there is any error, the *tftpLaunch()* function will return the error status code to the function which had invoked it.
- Launching the get file operation. It will invoke the *tftpGet()* function to achieve this. If there is any error, the *tftpLaunch()* function will return the error status code to the function which had invoked it.
- Launching the put file operation. It will invoke the *tftpPut()* function to achieve this. If there is any error, the *tftpLaunch()* function will return the error status code to the function which had invoked it.

The function calls the *Configure()* MTFTP4 protocol function and then, depending on the user's choice, it starts the get file or put file operations.

3.4.4 Get file operation

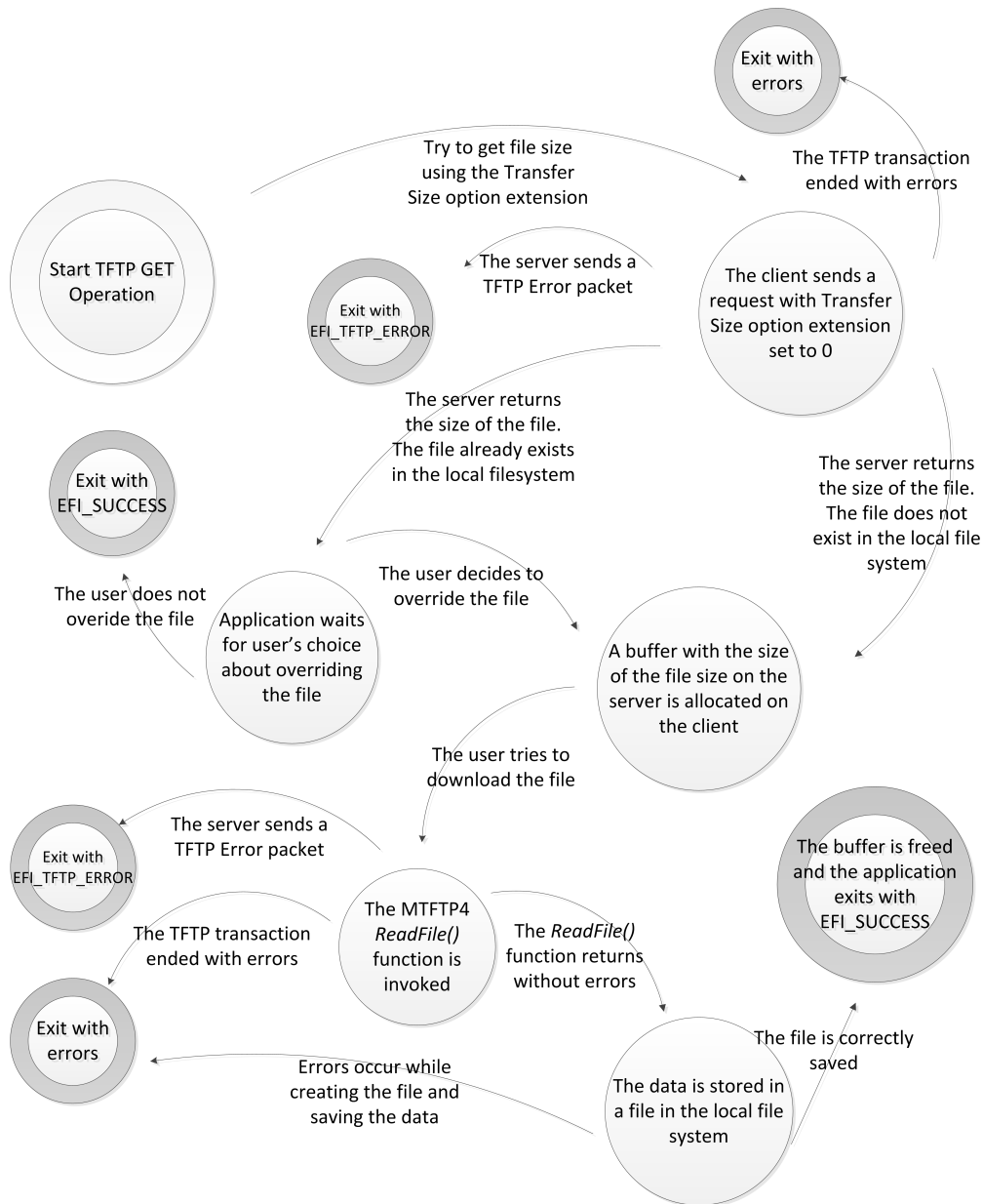


Figure 3.10: TFTP client get file operation

Figure 3.10 shows the TFTP get file operation. It starts when the *tftpLaunch()* function invokes the *tftpGet()* function.

tftpGet() and tftpGetFileSize()

These two functions are written on the *tftp* source code file. The *tftpGet()* function is invoked by the *tftpLaunch()* function. The function starts invoking the *tftpGetFile-*

Size() function in order to know how much memory it needs to allocate for the buffer which will be used to store the file data while it is being downloaded. The *tftpGetFileSize()* function invokes the *GetInfo()* UEFI MTFTP4 function. As explained in section 3.4.1, this function will return with an OACK TFTP packet sent by the server used to specify the options which it supports. Then, the *tftpGetFileSize()* function will use the *ParseOptions()* UEFI MTFTP4 function to extract the file size on the server using the transfer size option extension information of the OACK packet. If the server returns with a TFTP error packet an `EFI_TFTP_ERROR` status code will be returned to the user. The function ends freeing the resources which had been used and which are not needed anymore. This function can end successfully or with three different error codes:

- If it is an error status different from `EFI_NO_MAPPING` and it is not an `EFI_TFTP_ERROR` a generic error message will be printed on the standard output to inform the user.
- If the status is `EFI_TFTP_ERROR` it means that the server has sent a TFTP error packet. This information will be communicated to the user using an error message. The *tftpCheckPacket()* function will examine the TFTP error packet and it will provide the user more information about it (if the server provides this information).
- If the error status is `EFI_NO_MAPPING` no message will be printed because it will be the *tftpHandle()* function which will print it.

After that, the execution control will return to the *tftpGet()* function. Then the *existingFile()* helper function, written by the author of this report, will be invoked to check if the file which is going to be downloaded already exists on the local file system and it will communicate that to the user, giving him the choice to continue (or not) with the file transfer. If the file did not exist the process continues without interruptions. The buffer which is going to be used to download the file is allocated. Its size and a pointer to its position in memory are communicated to the MTFTP4 protocol using the Token structure.

Finally, the MTFTP4 *ReadFile()* function is invoked to perform the file download. It also returns a status code which is treated as the one returned by the *tftpGetFileSize()* function (in case of error). The file is downloaded in the buffer. To save the file in the current file system a helper function, the *writeFile()* helper function, written by the author of this report, function is used. If any problem occurs, it will return an error status code and the user will be notified about the impossibility of writing the file on the local file system. The function ends freeing the memory used by the buffer. If no error has occurred the user is informed about the file being

correctly downloaded. If there has been any problem, the proper error status code is returned.

3.4.5 Put file operation

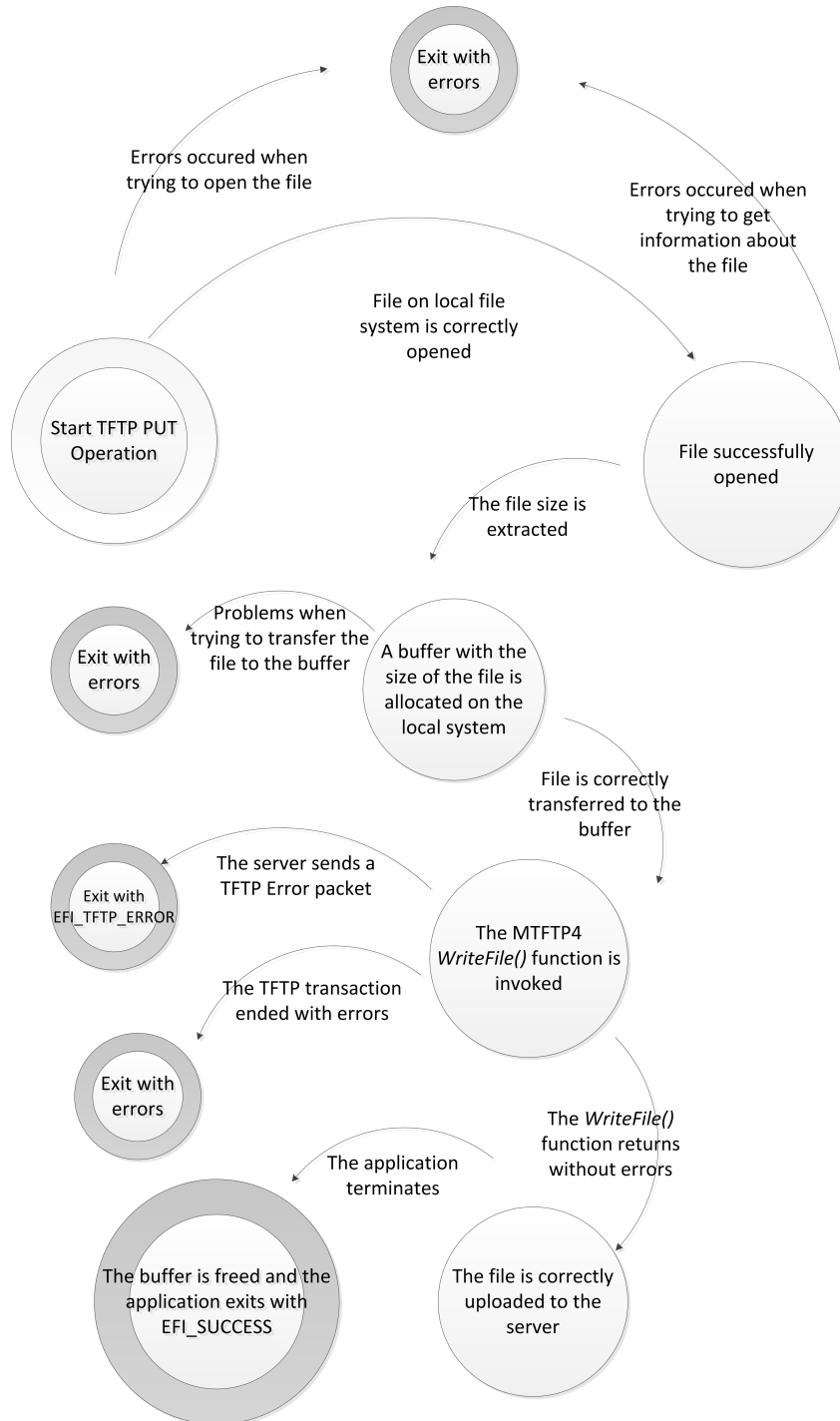


Figure 3.11: TFTP client put file operation

Figure 3.11 shows the TFTP put file operation. It starts with the *tftpLaunch()*

function invoking the *tftpPut()* function.

tftpPut()

First, the file which is going to be uploaded is opened. If an error occurs while opening it, it will be notified to the user. A helper function called *openFile()*, written by the author of this report, will be used to achieve this behavior. Then the file size will be stored in a variable. The information about the starting memory position where the buffer is stored and the file's size is passed to the MTFTP4 protocol using the Token data structure which was allocated by the *tftpHandle()* function. Then the *readFile()* helper function, written by the author of this report, is used to transfer the file from the file system to the buffer.

The UEFI MTFTP4 *WriteFile()* function is invoked to put the file stored in the buffer in the TFTP server. Finally the file is closed and the resources which are not needed anymore are released. The function also returns a status code which is treated as the one returned by the *tftpGetFileSize()* function (in case of error).

3.4.6 Help operation and TFTP error handling

Get line help operation

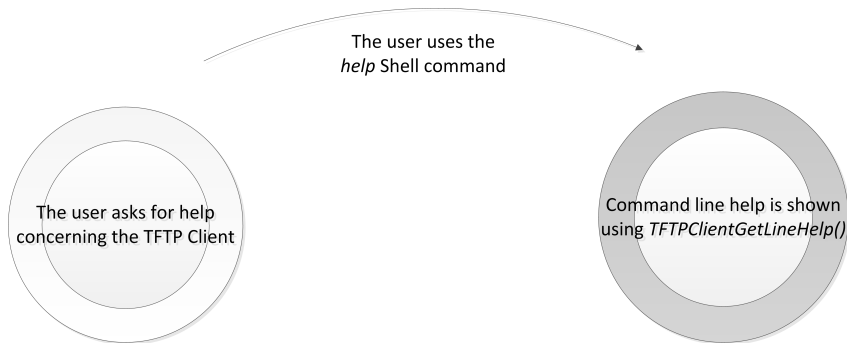


Figure 3.12: TFTP get line help operation

Figure 3.12 shows the TFTP client application get line help process. When the user uses the *help* command the *TFTPClientGetLineHelp()*, written by the author of this report in the TFTPClient source code file, is invoked.

Detecting TFTP errors - tftpCheckPacket() function

As it was said in section 3.3.4, the MTFTP4 Token structure needs a pointer to a function which scans TFTP packets searching for TFTP errors. This function is the *tftpCheckPacket()* function. It has been written by the author in the tftp source code file.

Although each time a TFTP error occurs the *tftpPut()* and *tftpGet()* functions will notify it to the user, the *tftpCheckPacket()* function will be used to extract more information about the error from the TFTP error packet (in case the server has provided this information) and it will communicate it to the user.

3.4.7 Helper functions

All these functions are written in the *helpers* source code file. They were written by the author of this master thesis report. They are used in the rest of the functions of the code to perform several operations.

UtoA10()

This function takes an integer variable and a memory pointer to an ASCII buffer as parameters, then it converts the number to an ASCII formatted string and stores it in the position pointed out by the buffer's memory pointer. It is used to convert the transfer size option, the block size option and the timeout option values from integer to ASCII, which is the format required by the MTFTP4 functions.

StrToIpT()

It is used to convert an Unicode string containing an IP address to the format used by the MTFTP4 to represent those addresses. The used data structure is the *EFI_IPv4_ADDRESS* structure, which is described in the UEFI specification [61]. This function also checks that the IP address is a valid IP address (although it does not check connectivity to the address). It is used to convert the server's IP address, provided by the user as a command line argument, to the required format.

UnicodeToAsciiT()

It is used to convert a Unicode string to an ASCII string. The application uses it to convert the filename of the file which is going to be downloaded from the server or uploaded to the server. The filename is stored as a Unicode string when the user provides it as a command line parameter and it is converted to the ASCII format required by the MTFTP4 protocol.

existingFile()

It is used to check if a file already exists before the client downloads another file with the same name from the server which would override it. The user is asked if the file should be overridden.

writeFile()

This function receives a filename and a buffer and its length as parameters and stores the buffer data on a new file which it creates. It is used when downloading files from the server to store the data, which was received in the buffer, in a file.

readFile()

This function receives a filename and a buffer and its length as parameters and stores the file data on the existing file. It is used when uploading files to the server to put the file data in a buffer and send that information to the server.

openFile()

It receives a file name as parameter and it tries to open it. It is used when uploading files, to check if the file which is going to be uploaded exists in the GEP board.

EfiAsciiStrnCmp()

It is used to compare ASCII strings. It is used when the server sends the client an OACK packet, in order to compare the packet's option names with the option names handled by the client, in order to extract the required option values.

AtoU64()

This function is used to convert an ASCII string to a 64-bit unsigned integer parameter. It is used to convert certain parameters inputted by the TFTP client's user from their ASCII format to an unsigned integer format.

3.4.8 Difficulties

The main difficulties which were experimented while developing the TFTP Client application were the following:

- Understanding the UEFI Shell environment of the EDK Shell's implementation: How UEFI applications are built, written, included in the shell's source code and executed. Due to the fact that the shell does not use the Standard C libraries it was necessary to study and put into practice all the concepts explained in the Shell Developer's guide [22].
- Being able to locate the MTFTP4 protocol instance handles was problematic. At first, it was only possible to use the TFTP client on the interface with the lowest Ethernet identifier. Nevertheless, using the *LocateHandleBuffer()* boot service the problem was solved.

- Accessing and changing the format of the arguments provided by the user in the command line. It can not be done as it is done using the C Standard libraries. Therefore, several previous applications had to be studied to design another method to solve this problem.
- Treating all the types of errors delivering all the proper error messages to the user of the application was a complex task due to the high number of possible errors.
- Many tests had to be run to find the proper default configuration for the TFTP client application in Ericsson's laboratory environment. It will be shown on the Analysis section of this chapter.

3.4.9 TFTP server application choice

The characteristics of the machine in which the TFTP server was set up are the following:

- Intel Xeon CPU L5518 @2,13GHz.
- 82575EB Gigabit Network Connection Card.
- SUSE Linux Enterprise Server 11 x86_64 Operating System.

The TFTP server application must have the following features:

- All their parameters must be easily customizable.
- It must be an open source server.
- It must support all the TFTP option extensions used by the client: timeout interval option extension, transfer size option extension and block size option extension.

Advanced TFTP Server

The chosen server software is the one included in the following package:

- Advanced TFTP Server and Client - Version 0.7.0-1.12

As it can be seen in its manual page [30] the *Advanced TFTP Server* has all the required features. It is an open source application and it supports all the required

options. It also supports one more option, the Multicast TFTP option defined in RFC 2090 [14], which could be an advantage if a Multicast TFTP service is required in the future. It has other useful features such as stats collection (the server logs the system load, the time between connections and other statistics such as number of files sent, received ...). It also allows to restrict the access to trusted hosts, which can increase the system's security level.

In order to install and setup this server on a *SUSE* environment several steps were followed:

- The Advanced TFTP Server and Client was downloaded and installed using SUSE's *YaST* utility.
- The server has a configuration file. It is a text file used to set some server parameters.
- One of the configuration parameters of the server is the folder in which the files are going to be read and written. This folder, and all its files and subfolders must have world-readable and world-writable permissions. The *chmod* Linux command can be used to set those permissions.
- The TFTP server can be run as a *daemon*. It can also be run using *inetd* which is a *super-server daemon*. It will run as a daemon as suggested by best practices from *openSUSE* [44]. Consequently, the TFTP server daemon must be restarted using the *rcatftpd restart* command.

The configuration file's parameters were modified to setup the server:

- The *ATFTPD_OPTIONS* parameter was set to *-daemon*. To setup the server as a daemon without using *inetd*.
- The *ATFTPD_USE_INETD* parameter was set to *no*. The server does not use *inetd*.
- The *ATFTPD_DIRECTORY* parameter was set to the desired directory.
- The *ATFTPD_BIND_ADDRESSES* parameter was set to an empty string. This parameter is used to setup the server interfaces which will be listening for new connections. It is set to an empty string because it is going to be listening on all its interfaces.

Other possible choices

Another possibility which also has all the required features is *Open TFTP Server* which can be downloaded from SourceForge web page ³. It also supports Windows environments. It was not the first choice because it is more difficult to configure and maintain than the *Advanced TFTP Server* and it offers the same features.

The *tftpd* utility [7] is another option which offers the same needed features as the *Advanced TFTP Server*. It is as easy to configure as the *Advanced TFTP Server*. It was as eligible as the *Advanced TFTP Server*.

The TFTP client application works only with servers implementing all the needed option extensions. The block size option extension is specially important to allow bigger file sizes. The transfer size option is even more important because it allows the client to use memory efficiently.

3.5 Analysis & Evaluation

These are the test environment's features:

- The network in which the client was tested consisted on two SUSE servers and several hosts. All the connections are Gigabit Ethernet connections. The network's MTU is 1500 bytes. The GEP board had 5 Gigabit Ethernet interfaces (but only four of them were used during the test because the fifth one is only used for debugging purposes).
- The *HxD - Hex editor* (which can be downloaded from its author web page ⁴) was used to generate several files with different file sizes. The contents of these files were randomly generated. According to the future users of the application, the maximum file size which will be used in the laboratory will be 100MB. Nevertheless nearly all the transferred files' sizes will be below 10MB.
- Using a USB drive, the files were transferred to the folder used by the Advanced TFTP Server to store its files.
- A GEP board with a 512MB RAM Disk drive was booted. The RAM Disk prototype (which will be explained in the following chapter) was used as the local storage drive for all the tests.
- A network analyzer, the *Ethereal network protocol analyzer version 0.10.14*, was used to analyze the output of some of the tests. It is installed on the TFTP server machine.

³<http://sourceforge.net/projects/tftp-server/>

⁴<http://mh-nexus.de/en/>

- A file comparator, *Beyond Compare Version 3.3.2*, was used to assure the files which were downloaded and uploaded were not corrupted. After a file was downloaded or uploaded to the server, it was transferred to a USB drive. Then the file was transferred from the USB drive to the laptop where the file comparator was run to assure that the byte contents of both files were exactly the same.

3.5.1 Command line arguments' tests

The first tests are designed to test how the application manages the command line arguments and the possible errors which may arise. All these tests are executed by downloading or uploading a 10MB file generated using HxD - Hex editor. The `get` command will be used to test the different parameters unless otherwise specified in the specific test. The file is checked with the file comparator after the file transfer has finished checking that it has not been corrupted. The tests and its results are described in the tables 3.1 and 3.2.

As it can be seen in those tables, the behavior of the client application when using the different arguments and flags is the expected one.

Command line arguments tests I			
Test	Action	Result of the file transfer	Comments
Get a 10MB file from the server	A get command without any flag was used	Success	The file was not corrupted
Put a 10MB file from the server	A put command without any flag was used	Success	The file was not corrupted
Use a non existing command different than put or get	The <i>scanfolder</i> command was used	Failure, the user is notified about the error	Only put or get commands are supported by the TFTP client
An IP address with an invalid format is used	The 300.300.300.245 IP server address is used	Failure, the user is notified about the error	The IP address must be a valid IP address
Use a duplicate flag	The port flag is used two times	Failure, the user is notified about the error	Flags cannot be used more than one time
Use a flag which does not exist	A non existing flag is used	Failure, the user is notified about the error	Only known flags can be used
The client is used with more or less than 4 arguments	The client is used with 3 and 5 arguments	Failure, the user is notified about the error	The TFTP client must be used with 4 mandatory arguments and with several optional flags
The port flag is used with a valid port which is in the 0-65535 range	The client is used with port 70. The server is modified to listen on port 70	Success	The network analyzer shows the port 70 was used
The port flag is used with an invalid port which is not in the 0-65535 range	The client is used with port 70000. The server is modified to listen on port 70000	Failure, the user is notified about the error	The server could not be started on port 70000. The client finished with an error message delivered to the user
The block size flag is used with a valid block size in the 8-65464 bytes range	A 1024 block size is used	Success	The network analyzer shows the block size was 1024 bytes
The block size flag is used with an invalid block size which is not in the 8-65464 bytes range	A 2 bytes and 70000 bytes block size are used	Failure, the user is notified about the error	The client finished with an error message delivered to the user

Table 3.1. Command line arguments tests I

Command line arguments tests II			
Test	Action	Result of the file transfer	Comments
The server timeout flag is used with a valid value between 0 and 255 seconds	A 15-second server time out is used. A 20-second client time out is used so the server time out is triggered first. In the middle of the download process the used Ethernet wire is disconnected from the board	Failure, the user is notified about the time out	15 seconds after disconnecting the wire the server sends a TFTP error packet with a time out error code
The server timeout flag is used with an invalid value which is not between 0 and 255 seconds	A 300-second server time out is used.	Failure, the user is notified about the incorrect parameter	The client finished with an error delivered to the user
The client time out flag is used with a valid value between 0 and 255 seconds	A 15-second client time out is used. A try count of 0 is specified. A 20 second-server time out is used so the client time out is triggered first. In the middle of the download process the used Ethernet wire is disconnected from the board	Failure, the user is notified about the time out	15 seconds after disconnecting the wire the user is notified about the error
The client timeout flag is used with an invalid value which is not between 0 and 255 seconds	A 300-second client time out is used.	Failure, the user is notified about the incorrect parameter	The client finished with an error delivered to the user
The try count flag is used with a valid positive integer value	A 15 second-client time out is used. A try count of 2 is specified. A 20 second-server time out is used so the client time out is triggered first. In the middle of the download process the used Ethernet wire is disconnected from the board	Failure, the user is notified about the timeout	45 seconds after disconnecting the wire (3 times the time out value) the user is notified about the error
The try count flag is used with an invalid value which is not a positive integer value	A try count of -3 is specified	Failure, the user is notified about the incorrect parameter	The client finished with an error delivered to the user
The Ethernet flag is used with a valid configured Ethernet interface included in the range of the 4 interfaces used by the board	Two Ethernet interfaces are connected to the server: Ethernet interface number 0 and Ethernet interface number 2. The interface flag is used to specify interface number 2	Success, Ethernet interface number 2 is used	The Ethernet 2 interface led which is activated when a certain interface is used is activated. The Ethernet 0 led is not activated
The Ethernet flag is not an unsigned integer	An Ethernet interface number of -3 is specified	Failure, the user is notified about the incorrect parameter	The client finished with an error delivered to the user

Table 3.2. Command line arguments tests II

3.5.2 TFTP-related tests

These tests are related to errors detected after all the arguments and flags have been validated. They can be seen in table 3.3. As it can be seen in the table, the behavior of the client application related to the possible TFTP errors is the correct one.

TFTP-related tests			
Test	Action	Result of the file transfer	Comments
Put file operation with a file with does not exist on the local file system	The user tries to put a file specifying a non existing local file name	Failure, the user is notified	The user is notified about the non existing file
Get file operation with a file with does not exist on the local file system	The user tries to get a file specifying a non existing remote file name	Failure, the user is notified	The user is notified about the non existing file by an error TFTP message which is sent by the server
Put or get file operation with a valid but non existing IP address	The user tries to get a file from a server which is not listening for incoming TFTP connections at the specified IP address	Failure, a time out event occurs and the user is notified	A time out will occur after waiting for a response to the read or write TFTP request.
A network interface which has not been previously configured using <i>ifconfig</i> is used to perform the file transfer	The user tries to use an Ethernet interface without previously configuring it	Failure, the user is notified	The user is advised to configure the interface
The user tries to put a file in a server without available storage space	The server's memory space is full with other files	Failure, the user is notified	The client receives a TFTP error message communicating that the server is full and it communicates it to the user
The user tries to put a file in a server which has no global read and write permissions on its TFTP folder	The TFTP folder's permissions are changed. The server administrator is the only user with access to the folder	Failure, the user is notified	The client receives a TFTP error message communicating that the server has denied the access to write or read its files
The user tries to get a file whose name is the same as the name of another file which already exists on its file system	An already existing file is tried to be downloaded	The client is asked for confirmation in order to override the file	The user can decide to override the file or to keep it

Table 3.3. TFTP-related tests

3.5.3 Timeout and try count tests

The following file sizes were tested: 20MB, 40MB, 60MB, 80MB and 100MB. Several time out values were tested: 0, 3, 6, 9, 12 and 18 seconds. The block size was 512 bytes (which is the RFC reference value). The try count was 0. Each value was tested 10 times with each file size. No time out event occurred.

Then the same test was repeated with the try count. The tested values were: 0, 1, 2, 3 and 4. The time out value was 0. The block size was 512 bytes. No time out event occurred. The reason for the absence of time outs is that there was no network congestion in the network. Due to the fact that Ericsson's laboratory network is a small network (less than 10 servers and hosts) and that there is a lot of available bandwidth in comparison with the average network load, a network congestion situation is really improbable. Nevertheless, the default try count has been set to 3. The default client time out value has been set to 2 seconds. These are conservative values to be able to deliver files even when there is some network congestion.

The author also wrote the TFTP client's application user manual, which is an internal document available to Ericsson. This user manual is described in the *Goals* section of this report and can be found in [39]. The application's user manual advises the user to use higher time out and try count values in an improbable situation with really high network congestion (the maximum recommended values are a try count of 4 and a time out value of 20 seconds).

3.5.4 Limit tests

The limit test cases are related to the size of the file which is going to be transferred. The case of a file with 0 bytes is tested. The theoretical maximum 4.29 gigabytes file size (derived from the maximum block size) is also tested. Nevertheless, GEP boards only have 1955 megabytes of total RAM available and it is necessary to allocate a RAM data buffer to store the data of a file while it is being downloaded or uploaded. Therefore, the maximum practical file size which can be downloaded or uploaded is 1955 megabytes.

In addition, as it will be explained in the RAM Disk chapter, the maximum file size when using the RAM Disk drive as the local storage device is 977 megabytes (limited by the GEP board's memory). Consequently, a 8-gigabyte USB external drive was used to execute the tests with the 1995-megabyte file.

The results are shown in table 3.4.

Limit tests			
Test	Action	Result of the file transfer	Comments
Get a 0 byte file	The user tries to get a file whose size is 0 bytes	Success	The file is downloaded
Get a 0 byte file already existing in the file system	The user tries to get a file whose size is 0 bytes and which already exists in the local file system	Success	The use can choose to override it or not
Put a 0 byte file	The user tries to put a file whose size is 0 bytes in the server	Success	The file is uploaded
Put a 0 byte file already existing in the server's file system	The user tries to put a file whose size is 0 bytes and which already existed in the server's file system	Failure, the server sends a TFTP error message	The Advanced TFTP server is configured to not allow overriding an existing file with another file with the same name and 0 bytes.
Get a file whose size is smaller than the transfer block size	The user tries to get a file whose size is 1 byte using a 512-byte block size	Success	The file is downloaded
Put a file whose size is smaller than the transfer block size	The user tries to put a file whose size is 1 byte using a 512-byte block size	Success	The file is downloaded
Get a file whose size is the theoretical maximum file size	The user tries to get a file whose size is 4.29 gigabytes	Failure	An error message notifies the user that there is not enough memory available to store the data file while it is being downloaded
Put a file whose size is the theoretical maximum file size	The user tries to put a file whose size is 4.29 gigabytes	Failure	An error message notifies the user that there is not enough memory available to store the data file while it is being uploaded
Get a file whose size is the maximum file size, 1955 megabytes	The user tries to get a file whose size is 1995 megabytes	Success	The file is downloaded
Put a file whose size is the maximum file size, 1955 megabytes	The user tries to put a file whose size is 1995 megabytes	Success	The file is uploaded

Table 3.4. Limit tests

3.5.5 Block size tests

Several tests were run in order to test the optimal TFTP block size in terms of throughput. A file whose size was 10 megabytes was randomly generated using HxD - Hex editor. A client time out value of 2 seconds and a try count value of 3 were used. Ten different block sizes were tested, starting with 512 bytes, which is the default block size and ending with 65464 bytes, which is the maximum possible block size using the block size option extension.

The 10-megabyte file was uploaded to the server ten times and downloaded from the server other ten times. A network analyzer was used to discover the time between the TFTP read or TFTP write request reception and the end of the transaction. The data is shown in appendix A. Then the average download and upload times were calculated for each block size. Finally, the average throughput for each of these block sizes was calculated. The results can be seen in figures 3.13 and 3.14.

As it can be seen, the throughput increases rapidly when increasing the block size. This result is supported by the results shown in the RFC 2348 [33] study. A similar study was performed in that RFC. Nevertheless, the block sizes range was smaller: 512 bytes to 8192 bytes. The test file size was also different, 2.25 megabytes. The explanation of this behavior is that increasing TFTP block sizes lead to a reduction of the packets which are sent. The number of data and acknowledgment packets is reduced by the same amount. Reducing the number of packets reduces the number of round trip times which take place during the transmission.

The time needed to upload or download a file depends on the network bandwidth and the number of round trip times. The network bandwidth can not be changed and does not depend on the client. The number of round trip times is decreased when the TFTP block size is increased. If the block size is increased so much that the TFTP messages exceed the path MTU, IP fragmentation and reassembly will occur and more overhead will be added. This overhead is only noticeable when the transmission between server and client must traverse several gateways and subnetworks. Nevertheless, the TFTP client and server are always on the same subnetwork. Therefore the effect is not noticeable. Consequently, the chosen default block size is the maximum block size, 65464 bytes.

In case gateways were added between the client and the server, the optimal block size will be the one which allows the biggest block size and at the same time does not cause IP fragmentation. This value is the Ethernet's MTU less the TFTP, UDP and IP header lengths. In this network the MTU is 1500 bytes, therefore the optimal block size is 1428 bytes in the case of a transmission traversing several gateways. The variances' variations between measurements, depicted in red, are due to slightly different network congestion levels during these measurements.

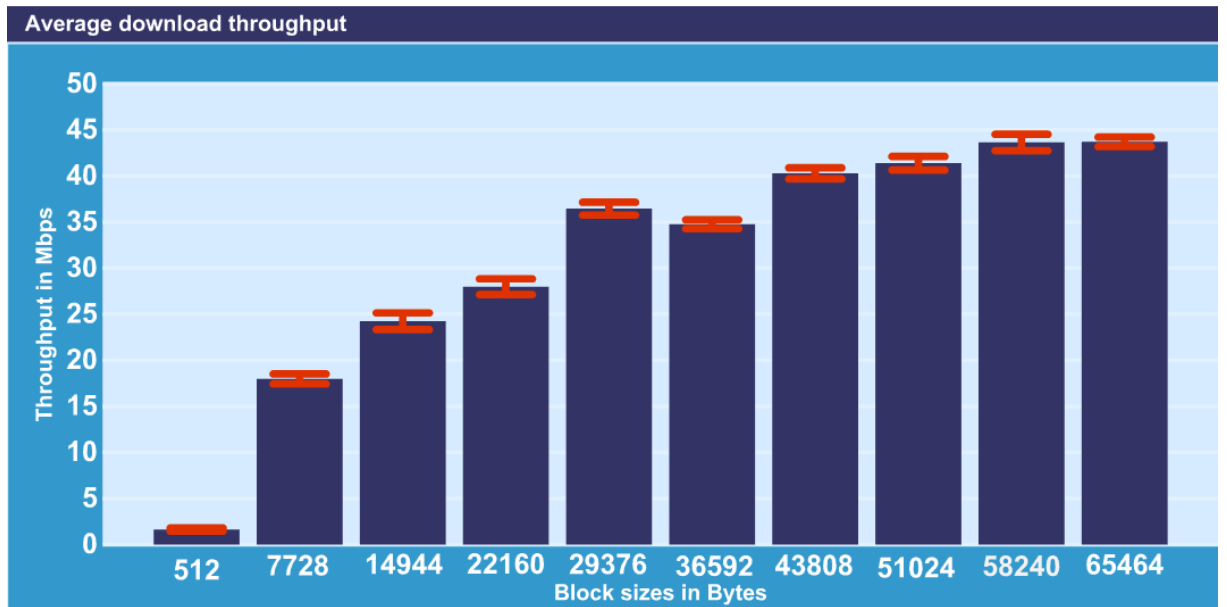


Figure 3.13: Average download throughput with variance

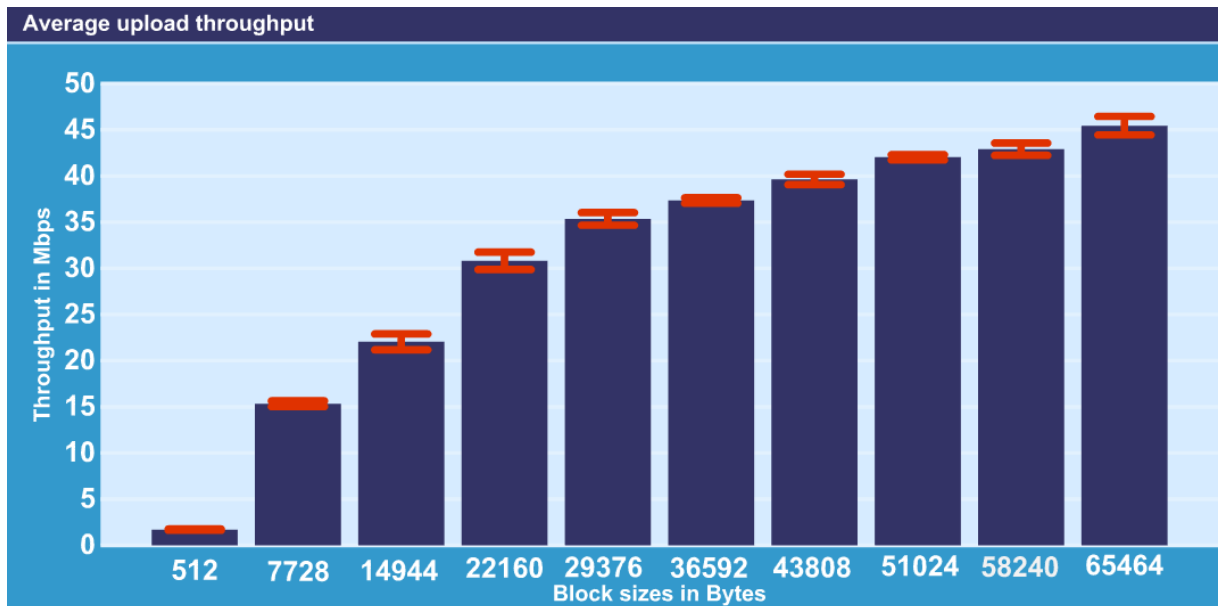


Figure 3.14: Average upload throughput with variance

3.5.6 Throughput tests

The TFTP client's upload and download throughputs were tested. The tests were executed with the client's default parameters and in a stable network state. In this case, a 50-megabyte file was randomly generated using HxD - Hex Editor. Then, the download and upload times were measured 50 times and the throughput was calculated. These data can be seen on appendix B. Finally, the data was analyzed using *R*, an statistical analysis software [2]. This software was used to extract and format the following information:

- Download and upload throughput comparison. It can be seen in figure 3.15. The bars depict the average throughputs. The vertical red segments traversing the bars represent the measurements' variances in Mbps squared.
- Download and upload throughput probability density functions, which represent the probability density of the occurrence of each of the throughputs. They can be seen on figures 3.16 and 3.17.
- The mean, variance and standard deviation of both throughputs. They can be seen on table 3.5.

As it can be seen in the download throughput density function, there is a peak around 44 Mbps. The upload throughput histogram and density functions present a main peak around 45 Mbps and a secondary one around 45.75 Mbps. Nevertheless, only one peak may be present around 45.5 Mbps with a higher number of measurements. The link between the client and the server is a Gigabit Ethernet 1000Base-T link and it is not the cause of the download and upload throughput limits. The client or the server network capabilities are not the cause of this limits. The cause of the limits is the nature of the TFTP connection. TFTP connections use a stop-and-wait Automatic Repeat Request (ARQ) protocol. This means that the server will not send the next TFTP packet until it has received the previous packet's ACK from the client. The maximum block size limits the minimum number of TFTP packets needed to transfer a file. Therefore the minimum number of round-trip delay times (which is the time it takes for a packet to be sent plus the time it takes to be acknowledged) is also limited by the maximum block size. The round-trip delay times are the most important components in the total amount of time needed to transfer a file using TFTP. The round-trip delay times limit the throughput.

Although the density functions are not Gaussian density functions, which is the expected theoretical result for these measures, they are similar to Gaussian ones. The differences with the Gaussian density functions are due to slightly different levels of network congestion during the tests. There was not a restriction for the

standard deviation and variance values for this test. The standard deviation is 111.25 times smaller than the mean in the download case. It is 92.65 times smaller than the mean in the upload case. By considering the means and standard deviations it is clear that both, TFTP write and read requests, have similar throughputs. Nevertheless, the upload throughput is a bit higher than the download throughput. This is due to the way the TFTP client application handles both types of requests. To handle read requests, the client triggers two TFTP transmissions. The first one is only used to get the file size on the server. The client closes that connection sending a TFTP error packet after receiving an OACK from the server with the file size. To handle upload requests only one transmission is triggered. Therefore, a download request will always require slightly more time than an upload requests. Consequently, the upload throughput is a bit higher than the download throughput.

Results			
Case	Mean	Standard deviation	Variance
Download	44.5 Mbps	0.4 Mbps	0.16 Mbps squared
Upload	45.4 Mbps	0.49 Mbps	0.24 Mbps squared

Table 3.5. Results

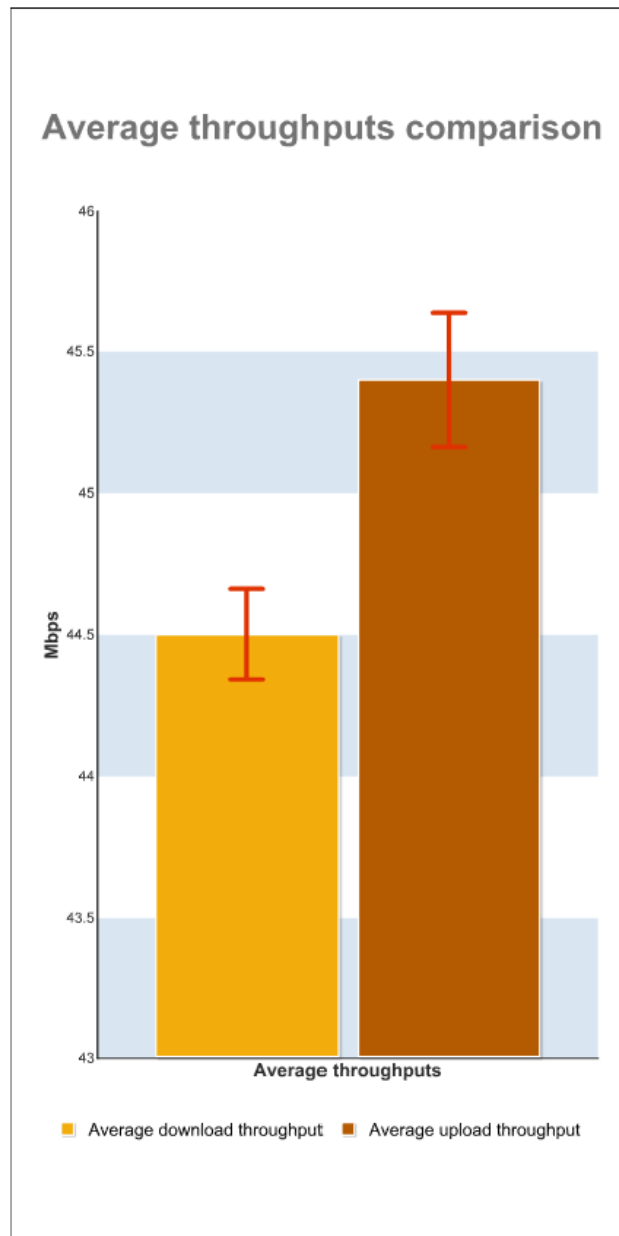


Figure 3.15: Average throughputs comparison with variance

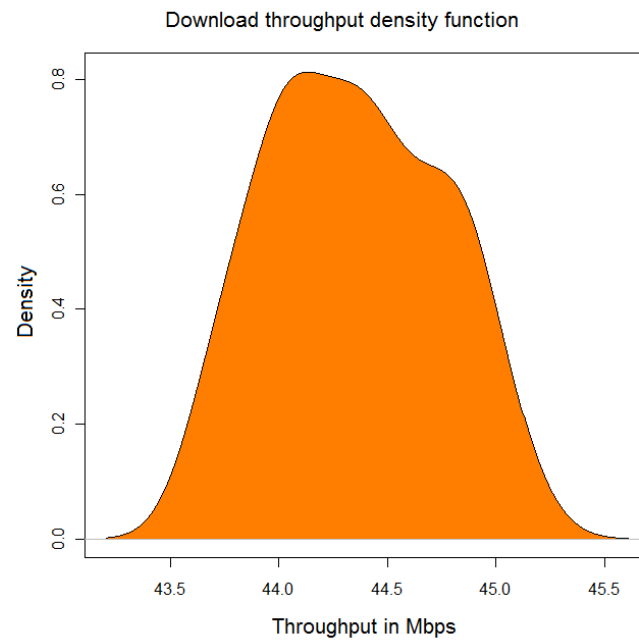


Figure 3.16: Download throughput density function

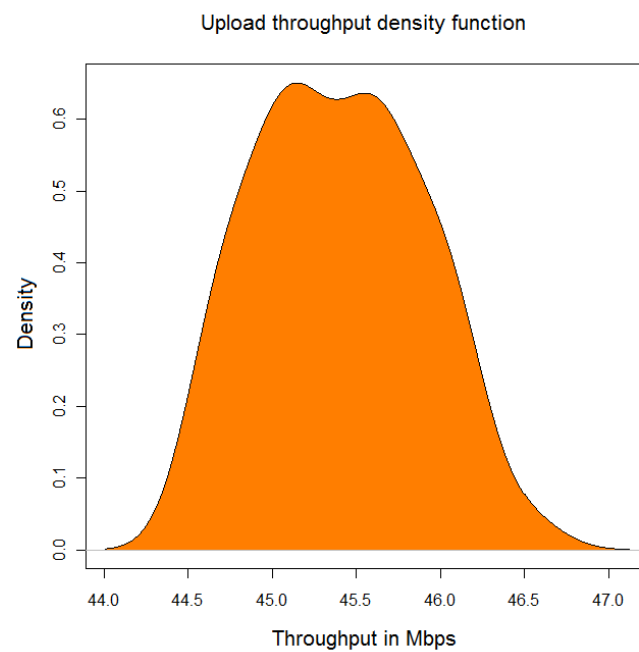


Figure 3.17: Upload throughput density function

3.6 Conclusions

This chapter has investigated the different possible implementations of a file transfer system for Ericsson's GEP boards. To achieve this goal FTP and TFTP protocols were studied and TFTP was chosen as the best option to build the required system due to the problem constraints.

A final prototype was designed, implemented and tested. The tests show that it fulfills the problem requirements.

- It is a UEFI Shell application which is ready for use after the GEP platform has been initialized.
- It does not need external peripherals.
- The throughput is acceptable. The average file size which will be downloaded or uploaded is around tens of megabytes, according to the users' information. Therefore, no more than two seconds will be needed to finish an average transmission, and they will usually be completed in less than one second according to the average throughputs.
- It reuses the MTFTP4 functions offered by the GEP board.
- Its memory size is 12 kilobytes, less than 1 megabyte.
- The error handling process and the error alerts presented to the user, in case an error occurs during the transmission, give the prototype the needed robustness to handle every possible situation.

3.7 Future work

The TFTP project functionalities can not be further improve. One way to improve the project would be to analyze the code efficiency and optimize it.

Also, another protocol should be implemented to improve the file transfer solution. One of the most important flaws of TFTP is its lack of security. It limits the prototype usage to secure environments. Therefore, if another prototype needs to be designed and implemented it should be a secure protocol.

Another problem is that TFTP does not allow to navigate through the server's folders, create folders on the server or change the name of files in the remote system. In other words, it is not an interactive protocol. If this kind of interactivity is needed in the future, and security is still not a concern, a FTP client application should be built.

The best way to implement a FTP client application for GEP platforms would be building a driver which implements the FTP protocol interface presented on the last UEFI specification [63]. Then, a FTP client shell application should be built using that driver. Implementing the driver first and then using it to support a shell application allows the driver's services to be available to other applications easily.

Chapter 4

RAM Disk drive prototype

This chapter describes the RAM Disk drive prototype. It starts explaining the motivation for a RAM Disk drive. Then it gives some background information about RAM Disk drives. It continues stating the prototype's requirements and the state of the art: available solutions and services. A first prototype is built using a RAM Disk driver included in the EFI Toolkit. The final RAM Disk drive prototype is also described. The chapter explains the prototype's design choices and implementation's difficulties. It also describes how it was tested and evaluated. Finally, some conclusions about the prototype and possible future work are commented.

4.1 Motivation and background

4.1.1 Motivation

As it can be shown in Appendix C there is a solid-state drive available at GEP boards. Nevertheless it is optional and it is not available in all GEP boards' installations. In addition, Ericsson prefers to use RAM memory-based storage systems during the pre-boot stage (when an OS has not been loaded yet). In addition, GEP boards do not need permanent storage during their pre-boot phase. The GEP boards only need volatile storage during their pre-boot phase, as a RAM-based solution provides. The RAM-based adopted solution was a RAM Disk drive.

4.1.2 RAM Disk drives

A RAM Disk drive is a virtual drive created in RAM memory which simulates a hard disk drive as shown by [46]. They are orders of magnitude faster than other secondary storage media as hard drives or solid-state drives, as shown by [27]. In addition, the maximum throughput of a RAM disk is limited by the speed of the RAM memory, the speed of the platform's processor and the RAM memory's data bus. Physical storage media has a stronger limitation due to the lower speeds of the storage buses which physical storage media employs, as shown by [6].

Disk file systems

Being a virtual disk, a disk file system should be mounted on it. These file systems take advantage of the ability to address data randomly, in a short amount of time, on a disk storage media, as shown by [50]. There are several disk file systems available. The most important ones are the NTFS file systems and the FAT file systems.

NTFS vs FAT file systems comparison

NTFS is much more advanced than FAT as shown by [41]. These are the advantages of NTFS over FAT:

- NTFS performance does not decrease as drive size increases.
- NTFS is more secure than FAT and it allows encryption.
- NTFS provides a system to monitor the amount of disk space used by each user called *disk quotas*.
- When a power failure occurs, NTFS provides logging information which allows quickly information restoration.

Sectors and clusters

Storage disks are divided in sectors which are subdivisions of tracks on the disk. Each sector can store a fixed amount of data (512 bytes in this project's final prototype). In order to reduce the overhead resulting from managing on-disk data structures, disk file systems do not allocate individual disk sectors, but continuous groups of sectors, called clusters as shown by [56]. It is the smallest logical amount of disk space which can be allocated to hold a file.

When allocating small files on a file system with large clusters some disk memory will be wasted. If the size of the cluster is small compared to the average file size, the wasted space per file is approximately the half of the cluster size. For large cluster sizes compared to the average file size, that wasted space will be greater. On the other hand, there is a problem related to disk fragmentation when using smaller cluster sizes. When a file is stored in a file system it is not always possible to store it in contiguous clusters, therefore different fragments of the file are stored in non-contiguous clusters. This is called disk fragmentation. When disk fragmentation occurs on a file system installed on a physical media with mechanical parts, the disk head needs to move to non-contiguous clusters when accessing to a fragmented file. Consequently, more time is needed to access files. Larger cluster sizes reduce fragmentation and may improve reading and writing speed on physical disks but due to the fact that the RAM disks are virtual disks, with no mechanical parts and a RAM-based media, it is not an advantage. Consequently, the cluster size should be chosen according to the average file size.

4.2 Requirements and state of the art

4.2.1 Requirements

The aim of this prototype is implementing a file system which is going to be mounted on a RAM Disk drive. These are its requirements for the GEP boards:

- The RAM Disk drive must be set up during the GEP platform initialization.
- Its contents should be accessible after the OS has been booted, if requested by the user.
- Its contents do not have to be accessible after a system reboot.
- It must have some mechanism to be loaded with different memory sizes on each system reboot.
- It must support all the UEFI Shell file system operations.
- It should be able to store at least 400 files.
- It should be built while the rest of the BIOS is being built instead of being built as an external driver. Therefore it will be faster to rebuild the system's BIOS when modifications to the driver are needed.
- It should not require more than 500 kilobytes of BIOS ROM memory to be loaded.

Due to the project's time limitations it was critical to discover if there was any available open source solution to solve this problem. It was also interesting to find out which available services might be used to build the foundation for a new application, in case the existing open source solutions did not completely fulfill the requirements.

4.2.2 Existing solutions

In order to discover if a RAM Disk drive open source project had already been developed the following sources of information were analyzed:

- Tianocore webpage: the website of the community which supports and develops the open source components of Intel's implementation of UEFI.
- EDK development email lists.
- Previous Intel EFI projects.

- EDK project's source code.

There was only one previous open source implementation of a RAM Disk drive on a UEFI pre-boot system. It was found in the EFI Toolkit and it will be explained on section 4.3.

4.2.3 Available services to build a RAM Disk drive

The UEFI Shell specification 2.1 [61] defines a protocol used to abstract mass storage devices so the code which is running in the EFI boot services environment can be used to access them without specific knowledge of the type of device or controller which manages the device. It defines several functions to read and write data at a block level from mass storage devices and it allows managing these devices in the EFI boot services environment. This protocol is the *BLOCK_IO_PROTOCOL* and it is implemented in the BIOS source code.

This protocol can be used to create a RAM Disk drive:

- A portion of the system's RAM memory must be allocated and defined as a mass storage device.
- Then, the *BLOCK_IO_PROTOCOL* protocol must be used to abstract it as an storage device.
- Finally, a file system should be mounted on the abstracted storage device.

Two types of file systems were considered:

- A non-standard file system using linked lists. It is a really simple file system in which each file has a memory pointer to the next file in the file system.
- Standard file systems, like *FAT16*, *FAT32* or *NTFS*.

The non-standard file systems are simpler to implement than the standard ones. They also provide all the needed functionality during pre-boot time. Nevertheless, one of the requisites of the RAM Disk drive file system is that it should be accessible after an OS has booted. Commercial OSs use standard file systems, therefore a standard well-known file system should be used. The *BLOCK_IO_PROTOCOL* protocol is used by the EFI Toolkit's RAM Disk drive. This RAM Disk drive is used to allocate a *FAT16* file system.

4.3 First prototype using the EFI Toolkit

The EFI Toolkit's RAM Disk drive has the following features:

- It provides a RAM Disk drive driver and a command line shell application which allows the user to load the RAM Disk drive specifying its memory size.
- The driver is built externally. Consequently it needs an external drive to be loaded.
- Its contents are not available after booting an OS.
- It supports all UEFI Shell file system operations.

As it can be seen, it does not comply with some of the requirements explained on section 4.2.1. The most critical one is that it requires an external drive to be loaded.

4.3.1 Adaptations to load the drive while the BIOS is being loaded

There are two possible solutions to the problem of including the EFI Toolkit RAM Disk drive on the BIOS source code:

- Building the driver externally using the EFI Toolkit environment and then including the binary image in the BIOS.
- Including the driver's source code in the BIOS and building it with the rest of the BIOS.

Although the first possibility does not comply with the RAM Disk drive requirements it is the simplest one. It is the simplest because it does not require modifying the driver's source code. It requires modifying the BIOS to include a new binary image. Although adapting the drive to include it as a binary image would not be a final solution, it could serve as the basis for the final solution. There are two possible solutions to the problem of including the Toolkit's driver as a binary image in the BIOS:

- Loading the RAM Disk driver binary using the BIOS source code.
- Loading the RAM Disk driver binary using the shell's source code, which is included in the BIOS source code.

Both possibilities offer the same advantages. Nevertheless, there was more documentation on the Tianocore community concerning the second option. Therefore the second option was chosen. The solution had to resolve two problems. The first one consisted on how the driver could be included as a binary image on the shell. The second one consisted in configuring the driver's automatic load process allowing the user to select a memory size for the RAM Disk.

Including the binary on the Shell

The first problem was solved including the driver's binary image as a C hexadecimal integer array in a header file. *HxD - Hexeditor* was used to analyze the driver's binary image. These were the steps which were followed:

- First, the driver's binary image was built using the EFI Toolkit environment.
- Then, the *HxD - Hexeditor* was used to copy its binary information to a text file. This binary information consisted on the binary code which would be loaded as a driver in the BIOS. It consisted on pairs of hexadecimal digits separated by whitespace characters.
- Windows 7 text editor was used to change the format of the sequence of pairs of hexadecimal digits to the one used in C to create an array of hexadecimal integers. This operation could have been done manually but it would have required a lot of time. It was faster to use the *Search and Replace* tool of Windows 7 text editor and regular expressions to complete this operation.
- Finally, an integer hexadecimal array was defined on a newly created C header file. This array contained the binary image information.

Configuring the driver's loading process

This process is explained in Figure 4.1. The shell's initialization code must be modified to load the binary image included in the header file. To do so, the source code file present in the shell's directory in the EDK project, *newshell/init.c*, must be modified. The function *InitializeShell()*, which can be found in that source code file was modified to invoke another function which was designed to load the driver specifying its memory size. This function is used to perform the same operations the EFI Toolkit's RAM Disk application performs. Nevertheless, the application's code can not be reused due to the fact that it will be executed in the shell initialization's environment, not in an external application.

The *EFI_LOADED_IMAGE_PROTOCOL* is used to select the driver's memory size. The *EFI_LOADED_IMAGE_PROTOCOL* is described on the UEFI Specification version 2.1 [61] and it is used to handle information about any image which is going to be loaded or which has been loaded on a platform. The user can store the

driver's desired size on a variable which will be loaded using the *LoadOptions* field. This variable could also be taken from other BIOS available memory structures.

Then, the *LoadImage()* boot service is used to load the binary image contained in the integer array, which was stored in a C header file, with the desired chosen driver's size. The *StartImage()* boot service is used to start executing the image. Finally, two shell commands described on [22] are executed using the *ShellExecute()* service. The UEFI Shell *map* command is executed to register the RAM Disk drive on the system. After executing the *map* command the drive is registered on the system. Nevertheless, it is necessary to also execute the shell *connect* command to order the system to make the RAM Disk available for the system's users.

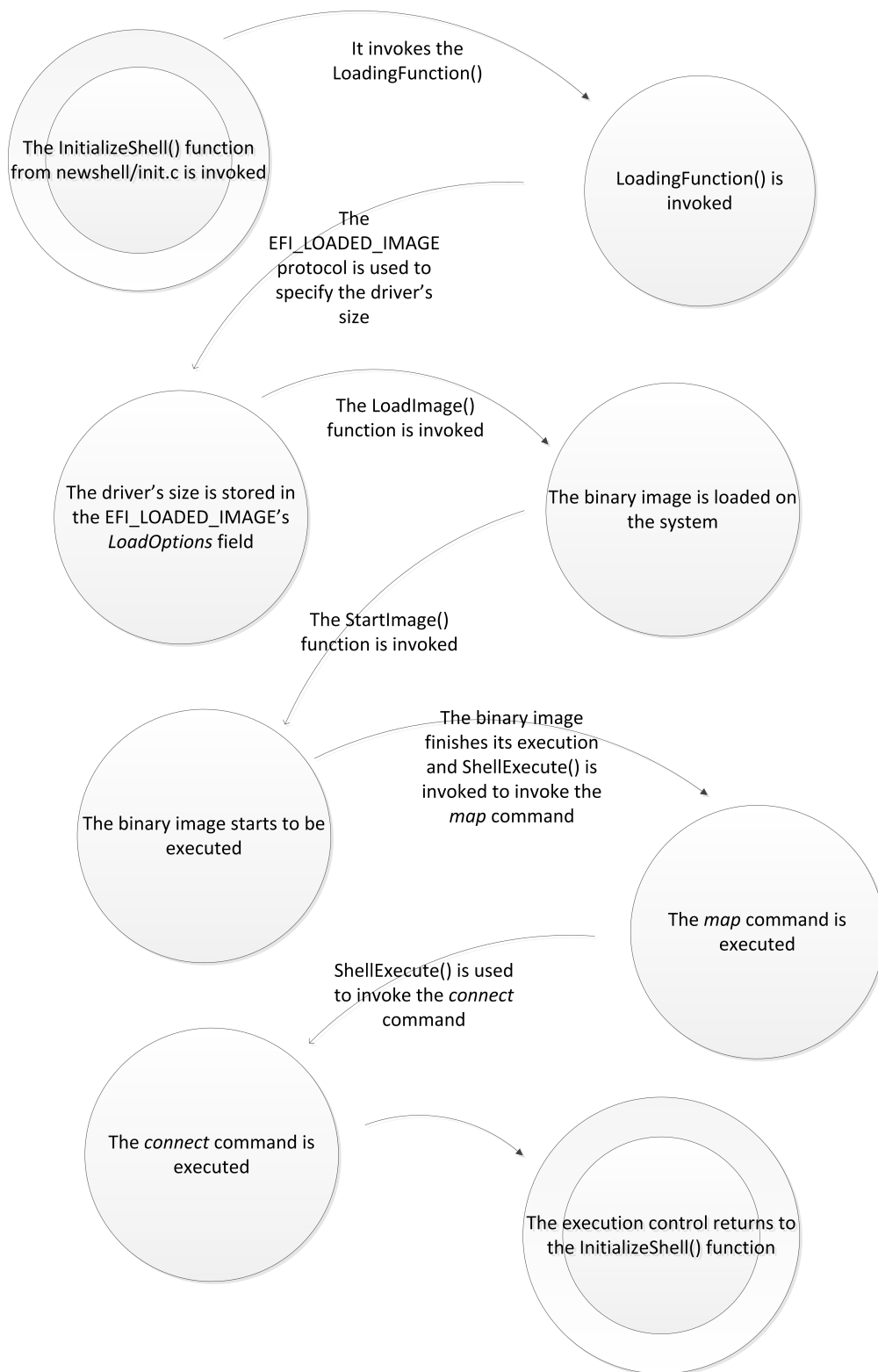


Figure 4.1: Configuring the driver's loading process

Reasons to explain why the first prototype is invalid

As it was said in section 4.3.1, this initial prototype was not a valid solution. It did not comply with two requisites:

- The RAM Disk contents were not available after booting an OS. This problem may have been easily solved as it will be explained in the final prototype section.
- The RAM Disk was built as an external driver. Although the resulting binary image could be included in the UEFI Shell and therefore in the BIOS, it required an independent building process. Therefore, if the source code of the driver needed to be modified, all the building process should have been repeated: building the driver using the EFI Toolkit environment; extracting the binary image information using a hex editor; formatting that information using a text editor and finally rebuilding the shell. This process requires several steps and too much time.

The RAM Disk driver's source code should be included in the BIOS' source code. Therefore, if the driver's source code needs to be modified, the process to build the new solution is easier and faster. Consequently the driver's source code should be modified and adapted to the BIOS environment in order to create the final prototype.

4.4 Final RAM Disk driver prototype design

The final RAM Disk driver is a modified version of the EFI Toolkit RAM Disk driver. In order to be able to modify the EFI Toolkit RAM Disk driver's source code it was necessary to understand how it was designed. This driver builds a RAM Disk drive using the UEFI *BLOCK_IO_PROTOCOL* and mounts a FAT16 file system on it.

4.4.1 Design choices - Disk file system

The first design choice is the selected disk file system. As it was said in section 4.1.2 NTFS and FAT are the eligible ones. The choice was motivated by the fact that they are supported by nearly every OS and one of the requirements of the RAM Disk drive is that its contents should be available after an OS has booted if the user requires it.

Although NTFS is more advanced than FAT, it is much more complex to implement NTFS than FAT. In addition, a FAT system is already implemented in the EFI Toolkit's RAM Disk drive. Although all the additional features and functionalities

offered by NTFS could be interesting in the future, they are not needed to solve the current problem. In addition, the project's time constraint requires selecting the simplest alternative which complies with the system requirements.

Among the possible FAT systems FAT32 or FAT16 could be chosen. FAT32 is superior to FAT16 as explained by [36]:

- FAT16 file system size is limited to 2 GB. FAT32 provides 2 TB of data.
- FAT16 file systems can only allocate 512 files. FAT32 does not have this limitation.

Nevertheless, as it will be explained later, the maximum RAM Disk drive is limited by the GEP board's RAM to 1955MB, therefore the first advantage is useless. The second advantage is also useless because, as explained on section 4.2.1, the RAM Disk drive needs only to be able to allocate up to 400 files. Finally, FAT16 is already implemented in the RAM Disk driver's source code.

4.4.2 Design choices - Cluster size

According to section 4.1.2, the cluster size must be chosen according to the average file size. In the case of the final prototype, due to the fact that there is not information available about the average file size of the files which will be stored in the RAM Disk, the cluster size is selected according to the total size of the RAM Disk. Therefore, if smaller RAM Disk sizes are allocated, the cluster size will be smaller and if bigger RAM Disk sizes are allocated, the cluster size will be bigger. This choice assumes that the user will allocate bigger RAM Disk drives when the average size of the files is bigger. This assumption might not be correct in all the cases, but it is reasonable. The size of the cluster will always be small (between 512 bytes and 32 kilobytes). This choice is motivated due to the fact that the fragmentation caused by choosing smaller cluster sizes will not impact the reading and writing overall speed because the virtual disk does not have mechanical parts, as explained on section 4.1.2.

4.4.3 Design choices - FAT16 boot sector

The boot sector is the very first sector in a partition. It has a lot of information which describes a FAT system, as explained by [64]. Some of its fields are:

- Number of bytes per sector.
- Number of sectors per cluster (which is used to specify the cluster's size).

- Number of sectors.
- Media descriptor of the device.

More information about this boot sector can be found on [64]. One static boot sector structure is declared in the driver's source code. Some of its fields are hard coded and others (like the number of sectors per cluster or the total number of sectors) are calculated dynamically when the driver is loaded, because the selected driver's size is needed to calculate them.

4.4.4 Design choices - BLOCK_IO protocol

The RAM Disk is implemented using this UEFI protocol, which is described in detail in [61]. It is used to abstract mass storage devices so the code which is running in the EFI boot services environment can be used to access them without specific knowledge of the type of device or controller which manages the device.

It defines several functions to read and write data at a block level from mass storage devices and it allows managing these devices in the EFI boot services environment. The data blocks are a logical abstraction of data used by the protocol. In the final prototype, the size of the logical block used by the protocol will be the size of a sector, 512 bytes. Therefore, UEFI logical blocks are equivalent to FAT16 sectors, which will ease the file system operations.

The protocol defines one data structure, the *EFI_BLOCK_IO_MEDIA* data structure, and four functions:

- *EFI_BLOCK_RESET*.
- *EFI_BLOCK_READ*.
- *EFI_BLOCK_WRITE*.
- *EFI_BLOCK_FLUSH*.

EFI_BLOCK_IO_MEDIA structure

It is a data structure associated with the block device. Its values are updated by the *EFI_BLOCK_IO_PROTOCOL* functions. These are its related fields:

- *MediaId*: It is the current media ID. Each block device which is managed using the *EFI_BLOCK_IO* protocol has an id associated to its media. If the media changes, this value will be changed.

- *RemovableMedia*: It is a boolean value to specify if the media is or it is not removable. In the case of the prototype, it is not removable.
- *MediaPresent*: It is a boolean value which specifies if there is some media currently present in the device. When the driver is loaded, this value is set to true.
- *LogicalPartition*: It is a boolean value which describes how the *EFI_BBLOCK_IO* protocol will be used. It can be used to abstract partition structures on the disk or to abstract logical blocks of data on a hardware device. The RAM Disk driver will abstract logical blocks.
- *ReadOnly*: It is a boolean value which specifies if the blocks on the device can only be read or if they can be read and written. In this case, they will be read and written.
- *WriteCaching*: A boolean value which is true if the protocol's *WriteBlocks()* function (which will be explained later) caches write data. In the case of the RAM Disk driver, the write data is cached.
- *BlockSize*: It is the block size (in bytes) of the device. In this case it is set to 512 bytes which is FAT16 sector's size.
- *LastBlock*: It is the last logical block address on the device. In the case of the RAM disk driver it is calculated as the integer division of the RAM disk driver size by the block size minus one.

EFI_BLOCK_RESET function

It is used to reset the block device hardware. It is not implemented by the RAM Disk driver. Instead of resetting the RAM disk, it should be formatted using a disk format utility like *diskpart* which can be found in the EFI Disk Utilities [17].

EFI_BLOCK_READ function

It needs the following arguments:

- A pointer to the *EFI_BLOCK_IO_PROTOCOL* instance.
- The id of the media in which the read operation has been requested.
- The starting logical block address to read from the device.
- A pointer to a data buffer where the read data will be stored and the buffer's size.

It is used to read blocks. All the requested blocks are read, or an error is returned. It is implemented in the RAM Disk source code.

EFI_BLOCK_WRITE function

It needs the following arguments:

- A pointer to the *EFI_BLOCK_IO_PROTOCOL* instance.
- The id of the media in which the write operation has been requested.
- The starting logical block address to write on the device.
- A pointer to a data buffer where the data which is going to be written to the device is stored and the buffer's size.

It is used to write blocks. All the requested blocks are written, or an error is returned. It is implemented in the RAM Disk source code.

EFI_BLOCK_FLUSH function

Its only argument is a pointer to the *EFI_BLOCK_IO_PROTOCOL* instance. It is implemented in the RAM Disk source code.

4.5 Final RAM Disk driver prototype implementation

4.5.1 Differences of the final prototype with the EFI Toolkit's driver

The final prototype uses the EFI Toolkit RAM Disk driver as a starting point. The mounted file system is a FAT16 file system. The driver is loaded with the rest of the components of the BIOS. Some messages about the status of the loading operation are printed in the standard output while it is loaded. The driver differs from the one which is present in the EFI Toolkit in several aspects:

- The Toolkit's driver is loaded using the functions present in the Toolkit's libraries. The libraries are a simpler version of the Standard C libraries. They also include some UEFI functions. It would be difficult to include all these libraries in the BIOS. It would not be an optimal solution. Consequently, a version of nearly all the library functions needed by the RAM Disk driver was searched and found in the GEP BIOS available libraries which are allocated in the GEP BIOS ROM. There were also some functions which were not found in the BIOS. Therefore they were ported from the Toolkit's libraries to the BIOS. Nevertheless, the number of these functions (as it will be shown in the following sections) is not big.

- The driver's loading process is also different. The Toolkit's driver can be loaded using the UEFI shell *load* command or the RAM disk loader application included in the EFI Toolkit, which allows the user to specify the RAM Disk drive's size using a command line argument. Nevertheless, the modified RAM Disk drive is designed to be able to be loaded without asking the user for command line arguments. Therefore, the default memory size of the RAM Disk is hard coded in the source code.

Nevertheless, as it is explained later, some functions were developed to be able to extract load options from external data structures during the BIOS boot process. These functions will be used in the future, when GEP platforms offer the needed mechanisms to do it.

- When the EFI Toolkit's driver is loaded, its device name in the shell is a long string extracted from the driver's device path. A modified version of the UEFI shell is provided with the RAM Disk driver which allows using a different, shorter and more comprehensible name for the RAM Disk drive.
- The Toolkit's driver was not designed to remain in memory after the OS is booted. The BIOS driver is designed to be able to remain in memory and it publishes the physical memory address where the driver is loaded and its size on the EFI system table and on an EFI variable, which can be accessed by a UNIX OS.
- The size of the EFI Toolkit's driver was limited to a range between 4.1MB and 512MB. The modified RAM Disk drive's size must be in a range starting in one megabyte and finishing in 1955 megabytes. One megabyte is the minimum file system size for FAT32 file systems, therefore it will be possible to format the file system and convert it to a FAT32 system if needed using the EFI Disk Utilities *efifmt* application [17]. 1955 megabytes is the maximum limit imposed by the platform's available memory.

4.5.2 Structure

The RAM Disk driver is composed by the file components explained in section 2.1.5. The driver's entry point function is specified in the Make file. The entry point is the *InitializeRamDiskDriver()* function.

As Figure 4.2 shows, the source code only consists on one file which contains the driver's entry point and the rest of the functions used by the driver during its load process. The code was not divided in several files because in this case the amount of code was not as big as in the TFTP client application case.

Appendix E describes a summary of Figure 4.2 functions and data structures. It states who is their author. It also states which other support routines and functions

Ramdisk
-RAM_DISK_DEVICE_PATH -RAM_DISK_DEV -RAM_DISK_POS -BOOTSEC -FAT16TABLE
+InitializeRamDiskDriver() +FormatRamdisk() +size2spc() +GetDiskSize() +RamDiskReadBlocks() +RamDiskWriteBlocks() +RamDiskFlushBlocks() +GetSize() +ReportSize() +GetMem() +ReportMem() +CopyMem() +AllocateZeroPool() +AllocatePool() +ZeroMem() +AtoiU() +MultU64x32()

Figure 4.2: RAM Disk drive structure

are invoked by the functions depicted in Figure 4.2. Finally it explains where these support functions are implemented in the GEP system.

All the RAM Disk drive prototype code is allocated as a GEP BIOS component and it is therefore stored in the GEP BIOS ROM. This RAM Disk prototype is based in the EFI Toolkit's RAM Disk source code [57]. Therefore its operation is performed using functions and data structures which were written in the EFI Toolkit's RAM Disk source code and other functions written by the author of this master thesis. There are four data structures which have been ported from the EFI Toolkit's code and one which has been entirely implemented by the author of this project.

There are two types of functions, the core operation functions and the helper functions. Some of the core operation functions are taken from the EFI Toolkit and they are not modified, others are taken and modified and finally there are functions entirely written by the author of this master thesis.

Some `EFI_BLOCK_IO_PROTOCOL` functions which are specified in the UEFI specification version 2.1 [61] are implemented as core functions. They are implemented by the EFI Toolkit's source code and they have not been modified.

The EFI Toolkit's RAM Disk also uses some helper functions which are written in

the EFI Toolkit's libraries. They support the RAM Disk core operation functions. Due to the fact that this master thesis RAM Disk prototype is based on the EFI Toolkit, the functionalities provided by the EFI Toolkit's libraries' helper functions are needed. To avoid porting the whole libraries to the GEP BIOS code, the GEP BIOS code was explored to find helper functions which offer the same functionalities as the ones provided in the EFI Toolkit's libraries. The GEP BIOS libraries have some of these functions, which are allocated in the GEP BIOS ROM. It was impossible to find all the needed functions in the GEP BIOS libraries to replace all the needed EFI Toolkit's helper functions. Therefore some EFI Toolkit helper functions were ported, without modifications, to the RAM Disk source code from the EFI Toolkit's helper libraries.

Finally some UEFI boot services and runtime services are used. They are not written in the RAM Disk source code. They are used to substitute some Toolkit libraries' helper functions but they are not part of the GEP BIOS libraries. They are part of the GEP BIOS drivers. They are implemented in the GEP BIOS source code and they are stored in the GEP BIOS ROM.

4.5.3 Code data structures

There are several data structures which are used to load the driver. More information about them can be found on appendix E.

RAM Disk device path structure

It is written in the Toolkit's driver source code and it has not been modified. It is used to create a UEFI device path for the RAM Disk driver in order to locate the driver in the system. One RAM Disk device path structure is declared in the source code. The name of the instance of this structure in the code is *RAM_DISK_DEVICE_PATH*.

RAM Disk device info structure

It is defined in the Toolkit's driver source code and it has not been modified. It has several fields. The ones which are relevant to the prototype description are explained here:

- Starting physical memory address: Obtained using the GEP BIOS helper *AllocatePages()* function when the RAM disk drive is allocated.
- Pointer to an *EFI_BLOCK_IO_MEDIA* data structure: Which is completed as explained in section 4.4.4.
- Pointer to an *EFI_BLOCK_IO_PROTOCOL* instance: The previous field is pointed by its *EFI_BLOCK_IO_MEDIA* data structure pointer. Its function

pointers (*EFI_BLOCK_READ*, *EFI_BLOCK_WRITE* and *EFI_BLOCK_FLUSH*) point to the EFI Toolkit's implementations of those functions (*RamDiskReadBlocks()*, *RamDiskWriteBlocks()* and *RamDiskFlushBlocks()*).

- Pointer to the RAM disk device data structure declared in the source code.

The name of the instance of this structure in the code is *RAM_DISK_DEV*.

RAM Disk position info structure

This structure is not present in the Toolkit's source code and it was written by the author of this project. It is used to store the starting memory position in which the RAM Disk is allocated and the RAM Disk's size. It will be used to pass this information to the OS, which could be configured to access the information stored in the RAM Disk which was used during pre-boot time.

Two fields describe the RAM Disk: the starting memory address, which is stored using 4 bytes and the RAM Disk size, which is stored using 2 bytes. The name of the instance of this structure in the code is *RAM_DISK_POS*.

FAT16 boot sector structure

It is defined in the Toolkit's driver source code and it has not been modified. It describes a FAT16 boot sector as explained in section 4.4.3. The name of the instance of this structure in the code is *BOOTSEC*.

FAT16 table structure

It is defined in the Toolkit's driver source code. It has been modified. It consists on the following elements:

- The total number of FAT16 sectors.
- The total number of sectors per cluster assigned to that total number of sectors.

It is used to select the number of sectors per cluster as explained in section 4.4.2. The Toolkit's version of this driver was modified because it could only be used with RAM Disks whose size was not bigger than 512MB. The final prototype does not limit the size of the RAM Disk with this structure. The name of the instance of this structure in the code is *FAT16TABLE*.

4.5.4 Loading process

All the functions described in this section are core operation functions. Figure 4.3 and Figure 4.4 show the RAM Disk driver loading process. The loading process is triggered by the BIOS loader when the BIOS is being initialized. Then the entry point *InitializeRamDiskDriver()* is executed.

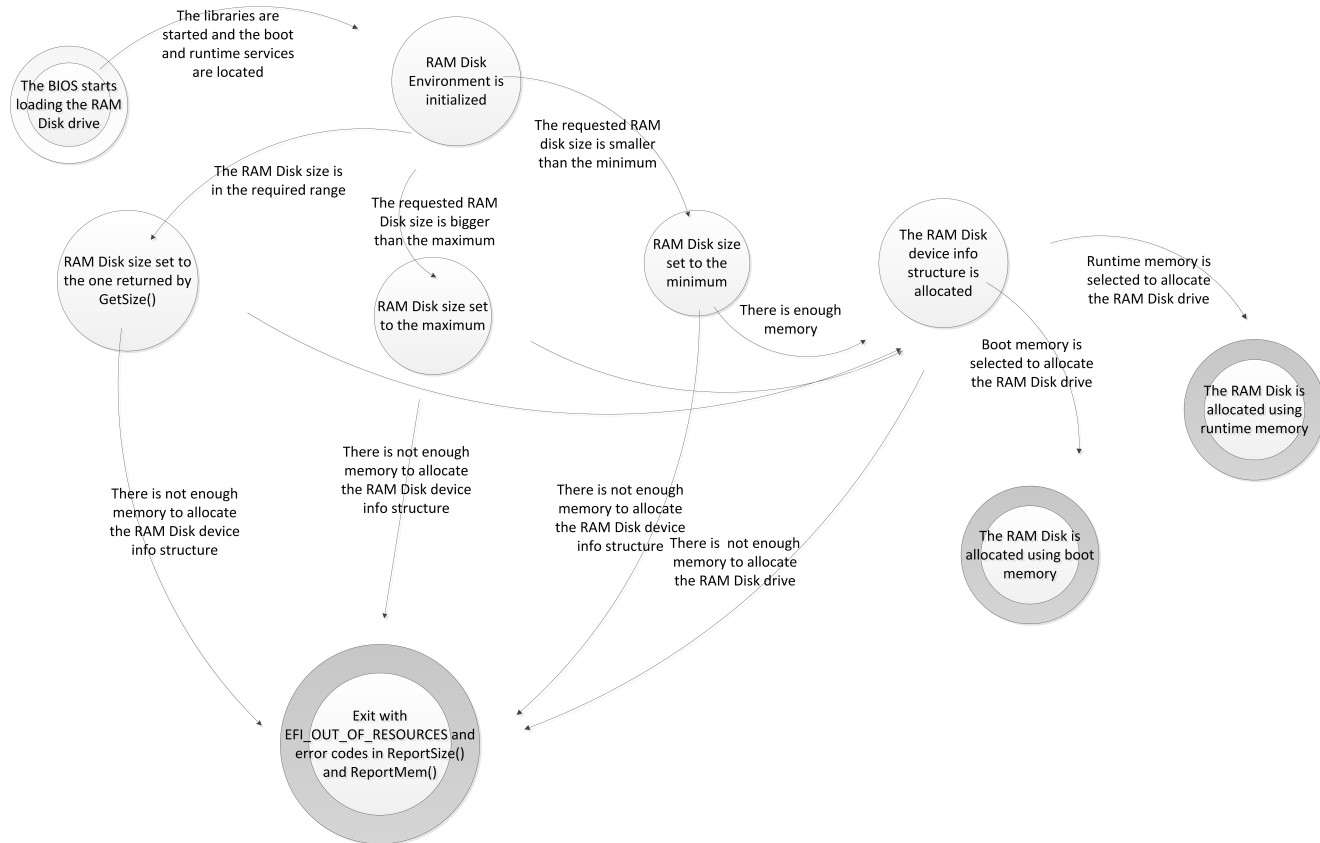


Figure 4.3: RAM Disk loading process I

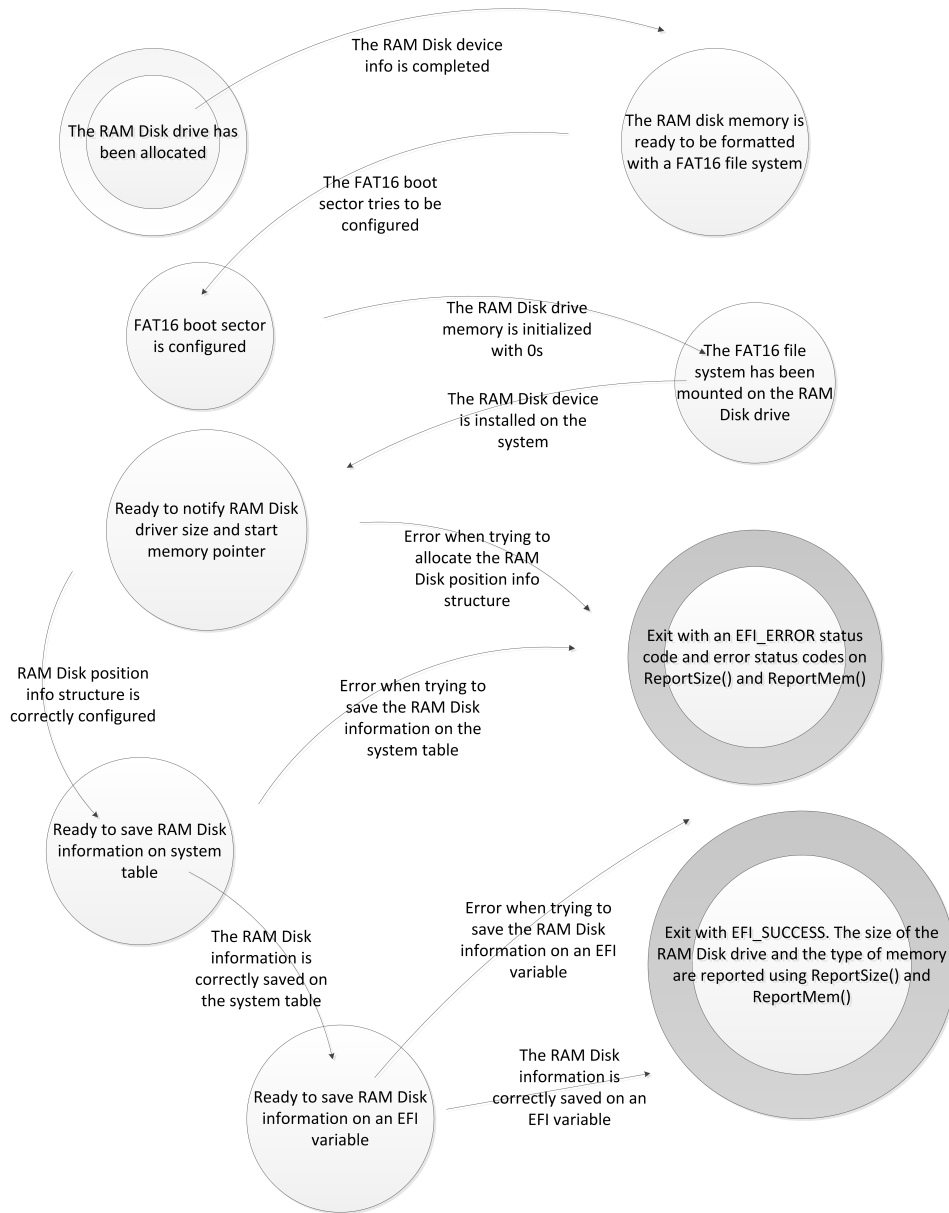


Figure 4.4: RAM Disk loading process II

InitializeRamDiskDriver()

The *InitializeRamDiskDriver()* function was already written in the EFI Toolkit's source code and it was modified. It starts initializing the GEP BIOS libraries which the driver needs and which are located in the BIOS ROM. Then it uses the system table pointer (a function parameter) to locate the boot services and runtime services function pointers. The *GetSize()* function, written by the author of this thesis, is used to get the desired RAM Disk drive size in megabytes. If the size is bigger than

the maximum allowed value it is set to that maximum allowed value. If the size is smaller than the minimum allowed value it is set to the minimum allowed value. Then the size is converted to bytes.

A RAM Disk device info structure is allocated. If there is not enough space to allocate it an *EFI_OUT_OF_RESOURCES* status code will be returned. In addition, the *ReportSize()* and *ReportMem()* functions, which were written by the author of this master thesis and which will be explained in the following sections, will return error codes. These two functions will always return an error code when a function returns with an *EFI_STATUS* error code. Then, the RAM Disk is allocated using the *AllocatePages()* GEP BIOS helper function. Depending on the value returned by the *GetMem()* function, which was written by the author of this thesis, the RAM Disk drive is allocated in runtime or boot memory (these types of memory are described in Figure 2.1), as it will be explained in the following sections.

The RAM Disk device info structure is completed as explained in section 4.5.3. Then, the *FormatDisk()* function, which was written in the EFI Toolkit, is invoked in order to initialize the RAM Disk drive memory and create a FAT16 file system in the allocated memory space. The device is installed on the system using the *InstallMultipleProtocolInterfaces()* UEFI boot service. Finally, some code was added to make some information about the RAM disk available to the OS.

The modifications to the EFI Toolkit *InitializeRamDiskDriver()* function were the following:

- The final prototype initializes libraries which are only present in the BIOS. The boot services and runtime services pointers need to be extracted from the system table (in the EFI Toolkit version they are directly available because the driver is designed to be directly loaded from the UEFI Shell, which provides these pointers).
- Then, as explained in section 4.5.1, to set the size of the RAM disk, the *GetSize()* function is used. The Toolkit version uses the function *GetDiskSize()*, which is used to extract load options, to set the RAM Disk driver's size from a UEFI Shell command line argument. Nevertheless, that function is not used in this prototype.
- Another difference is errors' treatment. The Toolkit's driver only returns a UEFI Status code when an error occurs. The BIOS version also uses the *ReportSize()* and *ReportMem()* functions to signal these errors. In addition, when the driver finishes loading with an *EFI_SUCCESS* status code, these functions are used to report the allocated size for the RAM Disk drive and the memory type in which it has been allocated, as it will be explained in the following subsections.

- Finally, some code was added to make the RAM Disk driver accessible after the OS has been booted. In order to accomplish this, some information about the RAM Disk is stored for the OS.

If the user wants to make the contents of the RAM Disk available for the OS, the RAM Disk drive must be allocated using the runtime memory type. This choice can be made using the *GetMem()* function. The process to store the RAM Disk information for the OS is the following:

- First, a RAM Disk position info structure is allocated. Then the size of the RAM disk and its starting memory position are stored on this structure. They will be stored in two different data structures.
- Then, the *InstallConfigurationTable()* UEFI boot service is used to store this information in the system table. More information about storing information in the UEFI system table can be found on section 6.5 in [61].
- The *SetVariable()* UEFI runtime service is used to store the same information in an EFI variable. An EFI variable is a variable which can be accessed by an UEFI system during pre-boot and which can be stored as another OS data structure when the OS is loaded. More information about storing information in EFI variables can be found on section 7.2 in [61].

Section 4.5.10 explains how this information could be used to access the RAM Disk drive if a UNIX SUSE OS is booted.

FormatRamdisk()

This function was written in the EFI source code and it has not been modified. It needs the memory pointer to the start of the RAM Disk drive and the size of the RAM disk provided as arguments. Given the block of memory representing a RAM disk, it builds a FAT16 pseudo-boot sector and initializes the drive. It assumes that the static boot sector structure has been filled with the static required information and completes the information which is left (the information which depends on the size of the RAM Disk drive size).

It computes the total number of sectors dividing the RAM disk total size by the number of bytes per sector. Then, it uses the *size2spc()* function, which was written in the EFI Toolkit, to calculate the number of sectors per cluster. After initializing all the fields it writes a 0 in all the RAM Disk memory bytes to initialize them. Then it copies the FAT16 boot sector to the start of the RAM Disk memory, occupying the first logical block.

GetDiskSize()

This function was written in the EFI source code and it has not been modified. It was used in previous source code versions to get the RAM Disk drive size using load options. It is not used in the last version but it remains on the code in case it is needed in future versions of the code. Its functionality has been replaced by the *GetSize()* function.

4.5.5 Read blocks operation

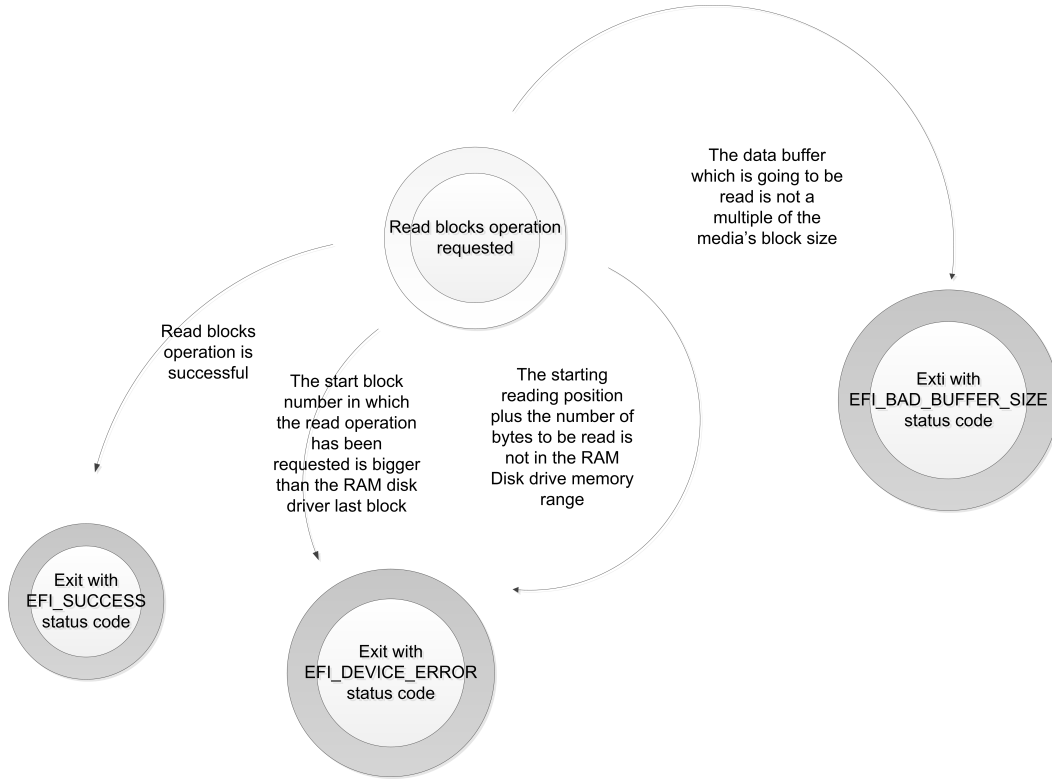


Figure 4.5: Read blocks operation

All the functions described in this section are core operation functions. Figure 4.5 explains the operation of reading blocks from the RAM Disk drive. It is implemented in the *RamDiskReadBlocks()* function.

RamDiskReadBlocks()

This function is an implementation of the *EFI_BLOCK_READ* function of the *EFI_BLOCK_IO_PROTOCOL*. It was present in the Toolkit's source code and it was not modified in the final prototype. A pointer to this function is attached to the pointer of the *EFI_BLOCK_IO_PROTOCOL* needed by the RAM Disk drive device info structure.

It returns *EFI_DEVICE_ERROR* when trying to read from a block number bigger than the drive's last block number. It returns the same error when the starting reading position plus the number of bytes to be read is out of the RAM Disk memory range. It also returns *EFI_BAD_BUFFER_SIZE* when the data buffer's size which is going to be read is not a multiple of the media's block size. If no error occurs, it reads the required data and ends with *EFI_SUCCESS*.

4.5.6 Write blocks operation

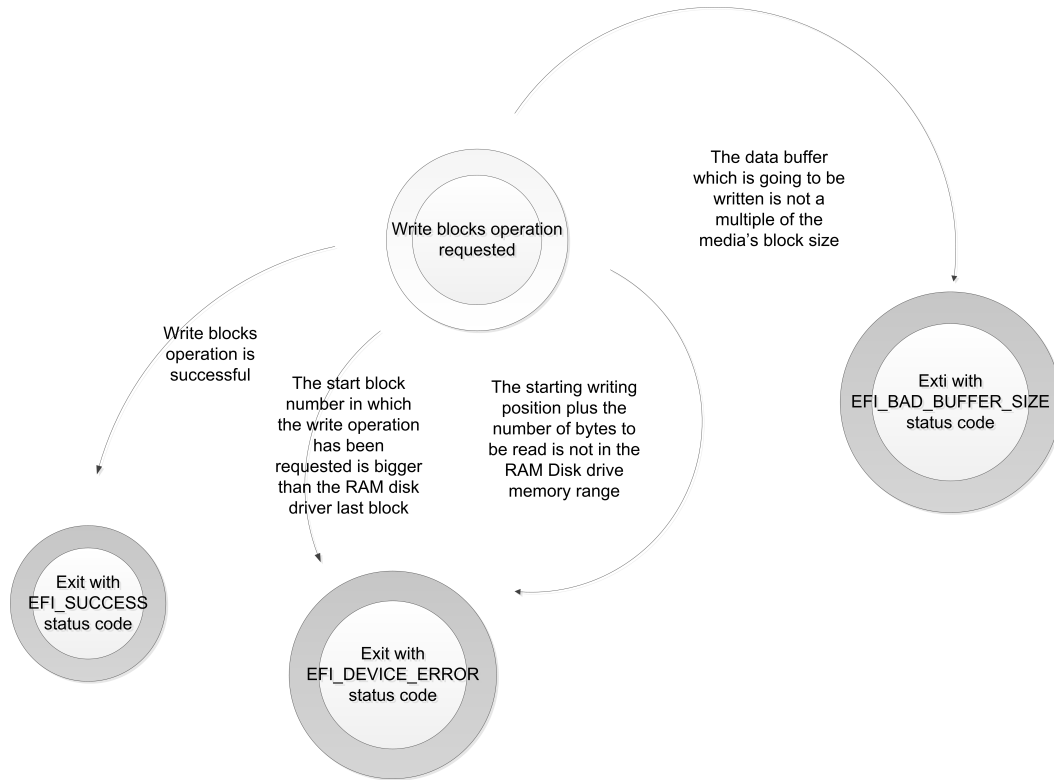


Figure 4.6: Write blocks operation

All the functions described in this section are core operation functions. Figure 4.6 explains the operation of writing blocks to the RAM Disk drive. It is implemented in the *RamDiskWriteBlocks()* function.

RamDiskWriteBlocks()

This function is an implementation of the *EFI_BLOCK_WRITE* function of the *EFI_BLOCK_IO_PROTOCOL*. It was present in the Toolkit's source code and it was not modified. A pointer to this function is attached to the pointer of the *EFI_BLOCK_IO_PROTOCOL* needed by the RAM Disk drive device info structure.

It returns *EFI_DEVICE_ERROR* when trying to write to a block number bigger than the driver's last block number. It returns the same error when the starting writing position plus the number of bytes to be written is out of the RAM Disk memory range. It also returns *EFI_BAD_BUFFER_SIZE* if the data buffer's size which is going to be written is not a multiple of the media's block size. If no error occurs, it writes the required data and ends with *EFI_SUCCESS*.

4.5.7 Flush blocks operation

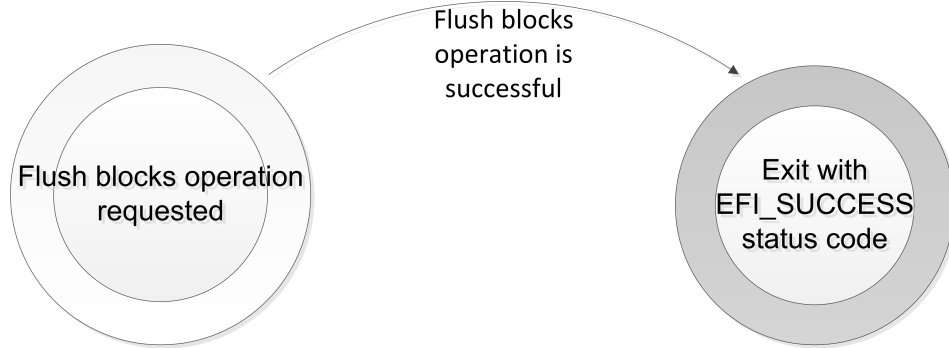


Figure 4.7: Flush blocks operation

All the functions described in this section are core operation functions. Figure 4.7 explains the operation of flushing blocks from the RAM Disk drive. It is implemented in the *RamDiskFlushBlocks()* function.

RamDiskFlushBlocks()

This function is an implementation of the *EFI_BLOCK_FLUSH* function defined in the *EFI_BLOCK_IO_PROTOCOL*. It was present in the Toolkit's source code and it was not modified. A pointer to this function is attached to the pointer of the *EFI_BLOCK_IO_PROTOCOL* needed by the RAM Disk drive device info structure. The implementation only returns *EFI_SUCCESS* when the flush function is invoked.

4.5.8 External parameters handler functions

All the functions described in this section are core operation functions and they were entirely written by the author of this project. The RAM Disk driver is loaded with the rest of the components of the BIOS. There are two driver parameters which can be set and changed:

- The RAM Disk size: It is a number between 1MB and 1955MB. The maximum recommended usable value is 977MB (half of all the available RAM memory after loading the BIOS). There is only 1955MB for both, the RAM disk driver and the TFTP client application's data buffers. Therefore, the maximum file size which can be downloaded or uploaded while using the RAM Disk driver is 977MB.
- The type of memory in which the RAM disk is allocated. It can be allocated in services memory or runtime services memory. The first one should be chosen in case the data stored in the RAM disk has to be deleted after loading the OS, which provides the OS more available RAM memory. The second one

must be used if the data stored in the RAM disk should remain accessible after booting the OS. The memory used by the RAM Disk drive can not be used by the OS's kernel (it is reserved memory).

There are four functions which are used to manage these variables in the source code:

- *GetSize()*: It returns the RAM disk size.
- *ReportSize()*: It reports the actual size of the disk after the allocation has been completed. A reported size of 0 means that an error has occurred.
- *GetMem()*: It returns 1 if boot services memory type has been chosen and 0 if runtime services memory type has been chosen.
- *ReportMem()*: It reports 1 for boot services memory type, 0 for runtime services memory type and -1 for error.

The *GetSize()* and *GetMem()* functions extract the desired RAM disk size and the desired memory type from some data structures before the system has been loaded. There exists no functionality in the GEP boards which allows extracting the desired parameters yet, but it may exist in the future. Therefore, in the current platforms, the RAM disk driver and the BIOS need to be rebuilt if the size or memory type values need to be changed because they are hard-coded.

The *ReportSize()* and *ReportMem()* functions are also not useful yet. They will be used to report the parameters which are being used or the errors that may have occurred while loading the RAM disk driver. They will be used when the GEP boards have the required functionalities to be able to write data to external data structures.

4.5.9 EFI Toolkit ported helper functions

These functions were written in the EFI Toolkit's helper libraries. They have been ported to the RAM Disk source code and they have not been modified. There were some functions from the Toolkit's driver libraries which could not be found on the BIOS code. Therefore they had to be rewritten and adapted to the driver's source code. They are only six functions. Therefore the size of the RAM Disk driver is kept small.

CopyMem()

Copies the memory specified by a length parameter from the source memory pointer to the destination memory pointer.

AllocateZeroPool()

It allocates a pool of memory whose size is specified as an argument and initializes it with 0s.

AllocatePool()

It allocates a pool of memory whose size is specified as an argument (without initializing it with any value).

ZeroMem()

It receives a portion of memory (described as a memory pointer and a memory size) as an argument and initializes it with 0s.

AtoiU()

It receives a Unicode string as an argument and converts it to an unsigned integer.

MultU64x32()

It is used to multiply a 64bit integer and a 32bit integer to get a 64bit result.

size2spc()

It used to calculate the number of sectors per cluster.

4.5.10 Using the RAM Disk information to access the RAM Disk when the OS has been booted

The information about the RAM Disk driver persists on the GEP OS in two data structures: the EFI system table and an EFI variable.

Accessing the system table information

The OS which is booted on the GEP boards is a *SUSE Enterprise Server Linux 2.6.39.1* OS. The `/sys/firmware/efi/` folder, in the OS, contains a text file which stores information about the EFI system table used during the pre-boot time. The RAM Disk drive source code adds information concerning the RAM Disk driver to this table. Some tests were run during the development of the RAM disk.

The result of these tests was that the system table was correctly stored in the system table text file. It contained several items. Nevertheless, the RAM disk information was not among them. The system table contains pointers to other configuration tables. Each configuration table has a GUID. Linux only handles standard configuration tables and the RAM Disk drive is a nonstandard (vendor-specific) configuration table. Therefore SUSE's kernel must be modified to detect

the RAM Disk GUID in order to be able to use the system table to extract the information.

Accessing the EFI Variable information

The `/sys/firmware/efi/vars` folder, in the OS, contains several folders which store binary files containing EFI variables which were set during pre-boot stage using the `SetVariable()` runtime service with the `EFI_VARIABLE_RUNTIME_ACCESS` flag set. The information about the RAM Disk is also stored this way. Several tests were run on the platform's OS and using the OS' hex editor the required information could be extracted.

Using the RAM Disk driver information to access the RAM Disk drive

This section provides some advice on how the RAM Disk information could be used to access the RAM Disk drive contents:

- The RAM Disk drive is still in memory after the OS has booted. The first step is to assure that the memory occupied by the RAM Disk driver (defined by a starting memory position and its size) is marked as reserved. To do so, the Linux kernel's `memmap` parameter can be used [13].
- Then, a new kernel module should be written. This kernel module should create a new block device to access the RAM Disk data. One sample application which is able to mount a file system specifying a start memory address is `pramfs` [55]. Although it can only mount a `pramfs` file system (and not a FAT16 file system), it can be used as a starting point for developing a FAT16 mounting module.

4.5.11 Changing the RAM Disk device name

The Appendix A of the UEFI Shell specification [62] specifies that all the mapped devices in a UEFI system must have a mapping name which is derived from the device's path in the system. The device path is used because it is unique in each platform. Therefore it is impossible to have two devices with the same device path. This implies that it is impossible to have two devices with the same mapping name. If the hardware configuration does not change, the device's path does not change.

After running some tests on the RAM Disk it was seen that its derived name was a really long string of characters without any meaning for the user. Some changes were made in the shell's source code to change the name of the device to `RamDisk0`. Therefore, the user is able to locate and use the RAM Disk more easily. These changes do not comply with the UEFI Shell specification. Nevertheless, the only possible risk would be connecting another device whose name, derived from its

unique device path, would also be *RamDisk0*. Due to the length of the device path variables, it is nearly statistically impossible that situation can occur.

The *ConsistMapping.c* source code file, which can be found on the *library* folder in the EDK shell's source code, was modified. Some lines were added to the *ConsistMappingGenMappingName()* function which are used to locate the RAM Disk device using its GUID. The *init.c* source code file, which can be found on the *new-shell* folder in the EDK shell's source code folder, was also modified. A new function called *ConfigRamDisk()* was created. This function, which is invoked from the *InitializeShell()* function, is used to perform the device name change when the shell is initialized.

The RAM Disk drive can be used with other UEFI shell implementations without modifications, using the name which the shell automatically generates for the RAM Disk using its device path.

4.5.12 Difficulties

These were the main difficulties experimented while designing and implementing the RAM Disk drive:

- Understanding the EFI Toolkit's driver code was necessary in order to modify it and create a new driver. It was a difficult task due to the lack of available code documentation. The code had to be analyzed and executed several times until its operation was understood.
- Understanding the GEP boards' BIOS environment was also hard because there was not documentation about it. It was necessary to understand how the BIOS loaded its drivers and components in order include a new driver.
- The GEP BIOS provides several libraries with general purpose functions. Nevertheless their code is not documented. These functions were used to perform the same functionalities as the ones provided by the EFI Toolkit's libraries. Finding these functions was a slow process. Due to the lack of documentation a lot of code had to be analyzed to find the needed functions. In addition, these functions' implementation was not exactly the same as the EFI Toolkit's implementations. Therefore they had to be tested to discover if their behavior was acceptable for the RAM Disk driver operation.
- In order to be able to change the name of the RAM Disk device, the EDK shell's source code was studied in great detail. The EDK shell's source code's size is big, and it was difficult to locate which functions had to be modified to change the device's name.

- Finally, there exists little information about designing pre-boot RAM Disk drives which remain available for the OS after the pre-boot stage is finished. The UEFI Specification [61] was analyzed in great detail, as well as SUSE's kernel documentation, in order to find a way to make the RAM Disk information available for the OS.

4.6 Analysis & Evaluation

These are the test environment's features:

- The machine in which the RAM Disk drive was tested was a GEP board. It had 1955 megabytes of available RAM before loading the RAM Disk drive.
- The board's OS' hex editor was used in some of the tests. The data was presented in a binary format.
- A USB drive was used to test some commands involving transferring files between the RAM Disk drive and another drive.
- The TFTP client application was also used in several tests.
- A file comparator, *Beyond Compare Version 3.3.2*, was used to assure the files which were transferred to/from the RAM disk were not corrupted (i.e. the byte contents of the original and the transferred files were similar).

4.6.1 Command line arguments' tests

The first tests are designed to test how the RAM Disk operates when the shell's file command line arguments, which are described in the UEFI Shell's Command Reference Manual [24], are used. Test files are needed for some of the tests. These files are generated using the HxD - Hex editor. All the tests which need file transferring between the RAM Disk drive and another drive are performed using a USB drive. If a test does not need a file transfer but requires a file to be present in the RAM Disk for executing the test it is assumed that the file was previously transferred from the USB drive and checked using the file comparator.

The files are checked with the file comparator after the file transfer has finished. The tests and its results are described in table 4.1. As it can be seen in the tables, the behavior of the RAM Disk driver when using different shell file commands is correct.

Shell's file commands tests			
Test	Action	Result	Comments
Show and change a file's attributes	The file attribute command is used to list a file's attributes and to change them	Success	The attributes are displayed
Change a directory	The change directory command is used to show the current directory and to move to another directory	Success	The directory is shown and changed
Copy a file	The copy command is used to copy a file from a RAM Disk directory to another. It is also used to copy a file from the USB drive to the RAM Disk drive	Success	The file comparator shows that the files are not corrupted
Delete a file	The delete command is used to delete a file from the RAM Disk drive	Success	The file is successfully deleted
Delete a directory	The delete command is used to delete a directory from the RAM Disk drive	Success	The directory is successfully deleted
Make a new directory	The make directory command is used to create a new directory	Success	The directory is successfully created
Move a file from one RAM Disk directory to another	The move file command is used to move the file	Success	The file is successfully moved
Change a file's date to the current time and date	The touch command is used to change the file's current time and date	Success	The time and date are successfully changed
Show the RAM Disk driver's volume information	The volume command is used to show the RAM Disk drive volume information	Success	The volume information is shown
List a directory	The list directory command is used to list a directory's files and its sub-directories	Success	The directory information is shown

Table 4.1. Shell's file commands tests

4.6.2 Limit tests

The limit test cases are related to the maximum and minimum sizes of the RAM Disk. The minimum size is 1 megabyte. The maximum size is 1955 megabytes. When the user requests a size which is bigger than the maximum size, the maximum size is chosen. When the user requests a size which is smaller than the minimum size, the minimum size is chosen.

There are also limitations concerning the files. Files whose size is bigger than the RAM Disk size can not be stored in the RAM Disk drive. The FAT16 file limit of 512 files is tested. In section 4.2.1 it is stated that the file system should be able to store up to 400 files. This case is also tested.

Results are shown in table 4.2. As it can be seen in the table, the behavior of the RAM Disk driver is the expected one.

Limit tests			
Test	Action	Result of the file transfer	Comments
A 1-megabyte RAM Disk size is selected	A 1-megabyte RAM Disk is tried to be created	Success	The RAM Disk is created and it occupies 1 megabyte of the platform's RAM memory
A 1955-megabyte RAM Disk size is selected	A 1955-megabyte RAM Disk is tried to be created	Success	The RAM Disk is created and it occupies 1955 megabytes of the platform's RAM memory
A RAM Disk whose size is smaller than 1 megabyte is selected	A 0.5-megabyte RAM Disk is tried to be created	Success	The RAM Disk is created and it occupies 1 megabyte of the platform's RAM memory
A RAM Disk whose size is bigger than 1955 megabytes is selected	A 2000-megabyte RAM Disk is tried to be created	Success	The RAM Disk is created and it occupies 1955 megabytes of the platform's RAM memory
A file bigger than the RAM Disk size is tried to be stored in it	A 1000-megabyte file is tried to be stored in a 512-megabyte RAM Disk	Failure	The file can not be stored on a RAM Disk which is smaller than the file size
600 files are tried to be copied from the USB drive to the RAM Disk	A shell script program is used to generate 600 files using the shell touch command and store them in the USB drive. Then they are tried to be copied to the RAM Disk drive	Failure	A FAT16 file system can only store up to 512 files. Therefore the first 512 file are copied, but the remaining 88 ones are not copied and an error message notifies it to the user
400 files are tried to be copied from the USB drive to the RAM Disk	A shell script program is used to generate 400 files using the shell touch command and store them in the USB drive. Then they are tried to be copied to the RAM Disk drive	Success	A FAT16 file system can store up to 512 files.

Table 4.2. Limit tests

4.6.3 Other tests

These tests are focused on different aspects of the RAM Disk drive:

- When the RAM Disk is used with the TFTP client application the file size limitation is 977 megabytes. This is due to the fact that the system has only 1955 megabytes of RAM memory available. Therefore, the maximum usable RAM Disk size is 977 megabytes, which is half of the available RAM memory. The other 977 megabytes can be used to allocate the data buffers for a 977-megabyte TFTP file which can be downloaded to the RAM Disk or uploaded from the RAM Disk to the TFTP server. If a 1000-megabyte RAM Disk is allocated, the maximum file size which could be stored in it due to the memory limitations would be 955 megabytes. These cases are tested.
- The RAM Disk is created using boot services memory and runtime memory. The shell *memmap* command is used to check that the RAM Disk is actually stored in the selected type of memory.
- The modifications done in the shell's source code to change the RAM Disk device name are also tested. The expected device name is *RamDisk0*.
- The information about the RAM Disk starting memory position and size is stored in the EFI system table, which is copied to an OS variable. As it was previously stated, this information is not accessible from the OS, its kernel has to be modified to recognize it. Nevertheless, the information stored on EFI variables is recognized by the OS, as shown by the tests executed using the HxD - Hex editor.

Results are shown in table 4.3. As it can be seen in the table, the behavior of the RAM Disk driver is the expected one.

Other tests			
Test	Action	Result of the file transfer	Comments
A 977-megabyte file is downloaded into a 977-megabyte RAM Disk	The file is tried to be downloaded	Success	The file is stored in the RAM Disk
A 977-megabyte file is downloaded into a 1000-megabyte RAM Disk	The file is tried to be downloaded	Failure	When the TFTP client tries to allocate a buffer to download the file, an error message notifies the user that there is not enough memory available to allocate the buffer
A RAM Disk is tried to be allocated using boot services memory	The RAM Disk is allocated	Success	Using the shell's <i>memmap</i> command it can be seen that the RAM Disk has been allocated in boot services memory
A RAM Disk is tried to be allocated using runtime services memory	The RAM Disk is allocated	Success	Using the shell's <i>memmap</i> command it can be seen that the RAM Disk has been allocated in runtime services memory
RAM Disk name test	The RAM Disk is allocated	Success	The shell's <i>map</i> command is used to check the RAM Disk driver's device name, which is <i>RamDisk0</i>
Accessing the information stored in the OS' EFI system table copy	The system table text file which can be found in the <i>/sys/firmware/efi/</i> folder in the platform's OS is analyzed using the OS' text editor	Failure	There is not information about the RAM Disk. The OS' kernel must be modified in order to save that information after the pre-boot stage
Accessing the information stored in the OS' EFI variables	The binary file which contains an EFI variable with information about the RAM Disk drive is analyzed using the OS' hex editor. The file is stored in the <i>/sys/firmware/efi/vars</i> folder	Success	The RAM Disk start physical address and the RAM Disk size are extracted from the binary information

Table 4.3. Other tests

4.7 Conclusions

This chapter has investigated the different possible implementations of a RAM Disk drive for Ericsson's GEP boards. An initial prototype involving several mechanisms to adapt EFI Toolkit's RAM Disk driver to GEP BIOS explored possible ways of solving the problem. A final prototype was designed, implemented and tested. The tests show that it fulfills the problem requirements perfectly.

- It is ready to use after the GEP platform has been initialized without requiring the user to perform additional operations.
- The starting physical memory address of the RAM Disk driver and its size are provided to the OS. Therefore, the OS can access its contents after booting.
- It provides several functions which can be used in the future to select the RAM Disk size and the memory type in which it will be allocated using BIOS external data structures.
- It supports all the UEFI Shell file system operations.
- It mounts a FAT16 file system which is able to store up to 512 files.
- Its source code is included with the rest of the BIOS code. Consequently, it is built when the rest of the BIOS is being built instead of being built externally.
- It only needs 45.5 kilobytes to be allocated on the BIOS ROM.
- It works perfectly with the TFTP client application to provide the user a Remote File Access System for GEP boards.

4.8 Future work

The RAM Disk driver can be improved using several approaches:

- It could implement other file systems. NTFS would be the best option due to its security features. These features could be used with a possible secure file transfer application to create a Secure Remote File Access System for GEP boards.
- The board's OS kernel should be analyzed and modified to use the data provided by the RAM Disk drive to be able to access it after booting.
- The GEP boards' memory resources should be upgraded in order to be able to allocate bigger RAM Disk drives.

- The current solution provides several functions to extract the RAM Disk size and the allocation memory type requested by the user from external data structures. These data structures should be designed and built on the GEP boards to take advantage of this feature.

Chapter 5

Secure file transfer protocols study

This chapter describes a secure file transport protocols study. It starts stating the motivation for the study. Then, the *TLS/SSL* and *SSH* protocols are analyzed and compared. Several secure file transfer protocols are described and their advantages and disadvantages are shown. The GEP BIOS resources which can be used to implement these protocols are presented. Finally, a comparison between all the possible options is shown and several conclusions are extracted.

5.1 Motivation

As it has been previously stated FTP and TFTP are not secure protocols. Their biggest security flaw is their lack of encryption mechanisms when transferring control and data messages. As a consequence, applications and drivers based on these two protocols can only be used in secure, trusted environments. Although a TFTP prototype has already been built for Ericsson's laboratory environment, Ericsson's future needs may include transferring data using insecure networks. Therefore a study which investigates the possible implementations of a secure file transfer system on GEP boards was performed.

5.2 TLS/SSL & SSH

TLS/SSL and SSH are used to provide secure file transfer services. They will be briefly presented and compared.

5.2.1 TLS/SSL

The Secure Sockets Layer, SSL, is a protocol which was developed by Netscape in order to use it to have secure HTTP connections. The IETF developed a standard based on SSL version 3.0 which was called Transport Layer Security, TLS. Several TLS RFCs have been written. The last one is RFC 5246 which describes TLS

version 1.2. The TLS protocol offers a full duplex byte stream which warrants privacy and integrity to its clients using public key cryptography techniques.

5.2.2 SSH

The Secure Shell protocol is a network protocol used for secure data communication. It is usually used to perform a remote login to a machine from a local machine over an insecure network. It uses public key cryptography and provides an encrypted terminal session with strong authentication capabilities. Several RFCs are used to describe this protocol: RFC 4254 for connection multiplexing, RFC 4252 for user authentication and RFC 4253 for encrypted data transport.

5.2.3 Comparison

Security

If both protocols are correctly implemented, both offer similar great levels of security. They use similar techniques to build a tunnel for confidential data transport with checked integrity, employing cryptographic methods. One difference is that SSH provides username/password authentication while TLS/SSL does not. Nevertheless, authentication methods can be built on the applications which use TLS/SSL.

Implementation cost

It is difficult to conclude which protocol is the easiest to implement. It depends on the protocol versions, the method of implementation and the available services.

TLS/SSL uses X.509 certificates for announcing server and client public keys, while SSH has its own format (SSH keys). SSH keys are easier to manage than X.509 certificates. This makes TLS/SSL a more burdensome protocol to use than SSH. In addition, a functioning public key infrastructure (*PKI*) is needed when using TLS/SSL. A PKI is a system used to create, manage and revoke digital certificates. On the other hand, a PKI allows scalable key management, while SSH does not.

Full TLS/SSL implementations are usually bigger and more difficult than SSH implementations. Nevertheless, if it is only partially implemented, TLS/SSL can be smaller than SSH. There exists embedded TLS/SSL libraries targeted for embedded systems (e. g. *axTLS* or *CyaSSL*). The following advices should be followed in order to develop an embedded, simpler TLS/SSL implementation:

- Use TLS 1.2, the latest version, which only requires implementing one hash function.

- Use only one cipher suite. Therefore, some values like the block size can be hardcoded.

Other features

These are other factors which should be considered:

- SSH is usually implemented with a set of protocols which manage the transfers inside the secure tunnel and add several functionalities: terminal management, multiplexing of several transfers or password-based authentication within the tunnel. TLS/SSL does not. Nevertheless, these capabilities can also be used with TLS/SSL, although they are not considered part of TLS/SSL. For example, when password-based HTTP authentication is executed over TLS/SSL, it is considered part of the *HTTPS* protocol.
- SSH and TLS/SSL are not usually used to solve the same types of problems. These differences are rooted in the different types of tools which historically came with the implementations of these protocols. SSH is usually used to perform authentication of remote systems. TLS/SSL is more used in online business operations. Nevertheless, both protocols can be used to solve the same types of problems.
- TLS/SSL is used as a general method for protecting data which is being transferred over a network. Therefore it allows implementing a wide range of applications. SSH is more focused on logging into remote computers. Nevertheless, there is also a wide range of applications which work using SSH.
- TLS/SSL uses X.509 certificates for public key management. These certificates include a public key and information about the certificate's owner. Therefore, the authenticity of the certificate owner can be checked. SSH keys contain only a public key without any information about the key's owner. Therefore SSH can not take advantage of the chains of trust which can be created using Public Key Infrastructures.

5.3 Secure file transfer protocols

5.3.1 FTPS

FTPS stands for FTP Secure and FTP-SSL. It is not a totally new protocol. It is the standard File Transfer Protocol with security extensions. These security extensions are based on TLS/SSL and are used to protect the control connection and the data connection. It has two operational modes: implicit mode and explicit mode. Nevertheless the implicit mode is deprecated. Therefore the following description is focused on the explicit mode. An explicit FTPS connection is similar to a FTP

connection but it allows the client to explicitly request security from a FTPS server, challenging it with a mutually agreed encryption method. The encryption method is negotiated between client and server before the challenge is sent.

According to its RFC [15], FTPS fully supports TLS/SSL cryptographic tools. Nevertheless, it is possible to implement a FTPS client which only uses certain ciphers and hash functions, to ease the implementation. On the other hand, this type of implementation would be able to communicate only with servers which implement that certain cipher and hash function.

Due to the fact that FTPS is FTP with security extensions, it has the same features as FTP. Therefore it can download and upload files, list directories, remove files or directories from the server or create new directories in the server.

Advantages

- The data and control connections' encryption can be disabled at any point during the communication between client and server. This may be useful for transferring files which are already encrypted at a file level.
- It is supported by many file transfer server software implementations as shown by [9].
- The protocol command messages use text format. Therefore they are human-readable when decrypted.

Disadvantages

- It needs a TCP control connection and a new TCP data connection for each file transfer. There are not usually firewall-related problems with the control connection. Nevertheless, it is common to experience them with the data connections. Some firewalls provide mechanisms to solve these problems.

5.3.2 SFTP

SFTP stands for SSH File Transfer Protocol and Secure FTP. It is a protocol which provides file and directory management services similar to the FTP ones. Nevertheless it is a totally different protocol. It is a secure protocol which can be implemented using several security schemes such as TLS/SSL or SSH. Nevertheless, it was designed as a SSH version 2 extension. Consequently it is almost always implemented using SSH. The protocol's design makes several assumptions such as:

- It assumes it is running over a secure channel.

- It assumes the server has authenticated the client before it starts working.
- It assumes the client's identity is available to the protocol.

The protocol is not an Internet Standard. Nonetheless, it is extensively implemented and there are several drafts [16].

Advantages

- It is supported by many file transfer server software implementations as shown by [9].
- A SFTP session needs only one connection. Consequently it does not suffer firewall-related problems.

Disadvantages

- The encryption can not be disabled at any moment. Therefore, files which are already encrypted at a file level will be encrypted twice, which is highly inefficient when transferring big files.
- The protocol command messages use binary format. Therefore they are not human-readable when decrypted.
- It is more complex to implement than SCP when SSH has already been implemented.

5.3.3 SCP

SCP stands for Secure Copy Protocol. It is a file transfer protocol implemented using SSH. It supports file transfers between a server and a client. Nevertheless, it does not support other functions such as directory listing or remote file removal. There exists no RFC which defines and describes the protocol.

Advantages

- The protocol command messages are a mixture of text and binary data, unlike SFTP which uses pure binary messages.
- A SCP session needs only one connection. Consequently it does not suffer firewall-related problems.
- It is easier to implement than SFTP when SSH has already been implemented.

Disadvantages

- The encryption can not be disabled at any moment. Therefore, files which are already encrypted at a file level will be encrypted twice, which is highly inefficient when transferring big files.
- It is supported by few file transfer sever software implementations as shown by [9].
- It does not provide support for directory listing or remote file removal functionalities as FTPS or SFTP do. It can only be used to transfer files from one host to another. As a consequence, many implementations use additional functionalities which are not described by the protocol, for example directory listing functionalities. Consequently, platform dependency problems are common.
- There is not an official description of this protocol. The available implementations' source code is the only documentation available. Nevertheless, it has been described by some developers as shown by [45].

5.3.4 Discarded options

There are two options which are not recommended:

FTP over SSH

It consists on tunneling the FTP protocol over a SSH connection. FTP uses a data and a control connection, which use different TCP ports. Consequently, it is difficult to tunnel this protocol over SSH. The control connection is easily tunneled over SSH. This control connection triggers the establishment of the data connection over another TCP port. The FTP over SSH client and server should be carefully designed to tunnel the new connection, which is usually established on a random port. It would be necessary to design a client and choose a server which monitor FTP connections.

In addition, as shown by [9], there are very few server software implementations supporting FTP over SSH. Other protocols such as *SCP* or *SFTP*, which also use SSH for security, are supported by more server software implementations and they do not suffer the problem of tunneling two different TCP connections over SSH.

Implicit FTPS

FTPS has two operational modes: explicit mode and implicit mode. In explicit mode the client can request the server to encrypt the FTP session. In implicit mode the session encryption is compulsory. Apart from the lack of flexibility which

this implies, this operational mode is not defined in FTPS RFC 4217 [15]. It was an earlier method of negotiating TLS/SSL encryption for FTP. Therefore it is now deprecated and it is advisable not to use it.

5.4 Existing resources

UEFI Specification 2.1 [61] provides basic support for encryption, hashing and signing capabilities, which are currently implemented in the GEP BIOS. The EDK project version 2 offers an open TLS/SSL ported library. It is not included in the GEP BIOS but it might be possible to adapt it to the BIOS environment. The library was included in EDK project version 2.

5.4.1 UEFI 2.1 capabilities already existing in the BIOS

EFI_AUTHENTICATION_INFO_PROTOCOL

This protocol is used to provide a *Secure boot* BIOS service. It allows to access generic authentication information associated with physical or logical devices. Therefore, the physical and logical components which are loaded during the BIOS loading process will only be loaded if the authentication information is correct. It can not be used directly to implement secure file transfer protocols because the protocol's authentication information is associated to devices instead of file transfer messages. Nevertheless, its source code could be adapted to create another protocol to authenticate messages instead of devices.

UEFI driver signing

UEFI executables can include digital signatures to verify them before they are loaded. This process is based on signing the driver using pairs of public/private keys. The driver signatures are embedded directly within the executable itself. The executable file's header contains a pointer to an entry with a list of certificates which may contain a digital signature used for validating the driver. In order to use a public/private key signature scheme some data is needed to be signed. In the case of the drivers the data is part of their executable file. In order to obtain the signature, the data is hashed using a hash algorithm. The signature is produced applying a private key encryption scheme on the hash value.

A structure named `WIN_CERTIFICATE_UEFI_PKCS1_15` is used to encapsulate the information needed to implement the *RSASSA-PKCS1-v1_5* digital signature algorithm specified in RFC 2437. This algorithm is used to sign drivers. Nevertheless its source code could be adapted to sign file transfer messages.

UEFI Hash protocols

The *EFI_HASH_PROTOCOL* can be used to generate and handle hash values for different messages. In this case, it can be used with any type of message. It does not have to be used only with drivers and logical or physical devices. It supports the following hashing algorithms: *SHA-1*, *SHA-224*, *SHA-256*, *SHA-384*, *SHA-512* and *MD5*.

5.4.2 EDK2 resources

EDK project version 2 is composed by several packages. As opposed to EDK version 1, these packages can be built separately without having to build the entire EDK project. One of these packages is the *CryptoPkg*. It is used to adapt the *Open SSL* library to the UEFI system. The adapted Open SSL library version is 0.9.8l.

The Open SSL library

The Open SSL library is an open source toolkit which implements the Secure Sockets Layer (SSL v2/v3) and the Transport Layer Security (*TLS* v1) protocols [43]. It is a robust, commercial-grade, full-featured toolkit. It also provides a full-strength general purpose cryptography library. The project also provides related documentation [43].

The CryptoPkg

The CryptoPkg package has the following components:

- Several static libraries. The Open SSL library is one of them. Its source code is not modified in the package. Nevertheless, it is adapted to the UEFI system using other libraries.
- A runtime driver. It has several wrapper functions which allow using the Open SSL library.
- An application to test the package.

Advice on including the CryptoPkg in the BIOS

In the UEFI environment each application or driver has its own copies of the libraries that it uses. There is not dynamic linking in UEFI, everything is abstracted via protocols. Therefore, the problem when porting the CryptoPkg to the GEP BIOS are the protocols consumed by some library functions. Nevertheless, the protocols consumed by the CryptoPkg package are already built in GEP BIOS. The CryptoPkg output are mainly pure static libraries and a runtime driver. In order to include this package in the BIOS, the package should be built using the EDK

2 environment. Then, the driver's binary image should be included in the same binary image as the GEP BIOS.

Once the driver has been included into the BIOS it can be used by other applications and drivers. These applications should be built using a modified version of the EDK 1 environment:

- The CryptoPkg package must be built using the EDK 2 environment.
- Then, the corresponding library files (*.lib* extension) and header files should be included into the EDK 1 environment. The following libraries should be included: *BaseCryptLib*, *OpensslLib* and *IntrinsicLib*.
- Nevertheless, the porting process may generate some difficulties. The BaseCryptLib library uses some functions from other EDK 2 libraries. Therefore it would be necessary to port these functions or adapt them to EDK 1 environment.

In order to elaborate this description on how to port the CryptoPkg to GEP platforms several sources of information were analyzed:

- EDK 2 project general structure.
- EDK 1 project general structure.
- CryptoPkg package source code and components.
- EDK developers' email lists.

5.4.3 Porting other resources

SSH is also used by some secure file transfer protocols. Nevertheless, there is not support for this protocol on any implementation of the UEFI specification. There are two solutions to this problem:

- Write a new SSH driver using the available functions and protocols offered by the GEP BIOS code.
- Port SSH code to the GEP BIOS code.

Advice on porting SSH code to the BIOS

Instead of writing a new SSH driver it could be faster to port already existing SSH code to the BIOS. The CryptoPkg package should be studied in order to discover how the Open SSL library was ported to the EDK 2 environment. Then, the same mechanisms could be used to port a SSH open source implementation to the EDK 1 environment. These are the two possible options:

OpenSSH It is a set of tools based on SSH which include two secure file transfer protocol client implementations: *SCP* and *SFTP* [42]. It is not a library. Therefore it would only be necessary to port the desired secure file transfer protocol client applications.

LibSSH It is a C library which implements SSH protocol version 1 and version 2 [31].

Nevertheless this process would be harder and slower than using the CryptoPkg package. In addition, the EDK developers' community stated on their email lists that it would be possible to port the CryptoPkg package from the EDK 2 environment to the EDK 1 environment. Nevertheless, there are not warranties on the possibility of porting an open source SSH library to the EDK 1 environment.

5.5 Comparison

5.5.1 Discussion

Table 5.1 summarizes the possible choices. The optimal protocol choice depends on the secure file transfer application desired functionalities. The first parameter to take into account is feasibility. FTPS works on top of TLS/SSL while SFTP (typically) and SCP work on top of SSH. As stated in section 5.2, TLS/SSL is generally more complex to implement than SSH. Nevertheless, there exists UEFI resources which could allow to implement TLS/SSL faster than SSH.

All the protocols offer good levels of security. Only FTPS uses human-readable command messages (after decryption). Nevertheless, this feature is transparent for the user. Although firewall problems were critical when FTPS started to be deployed they are not critical any more: servers offering FTPS features have their firewalls properly configured to avoid these problems. Therefore, if the client configures its network firewall properly, there should not be any firewall-related problems.

The possibility to disable encryption is not critical unless the majority of the transferred files are already encrypted (in which case a secure file transfer protocol would only be needed for transferring control messages). The number of commercial servers

implementing each protocol is an important parameter. The reason to develop a secure file transfer application is to be able to communicate with servers which are outside Ericsson's secure environment. Therefore, it is better to implement a protocol which is supported by a great number of server software implementations. SCP would not be a good choice according to this parameter.

The amount and quality of documentation is not critical itself. Nevertheless, the lack of SCP specifications has provoked some incompatibilities between different implementations. Finally, both FTPS and SFTP offer file system-related commands, such as directory listing, the possibility to create new directories in the remote host or remove files or directories. SCP only allows to securely copy a file from one host to another.

Comparison			
	FTPS	SFTP	SCP
Underlying protocol	TLS/SSL	Usually SSH (but can be used with others)	SSH
UEFI Support	Ported CryptoPkg package	No (if using SSH)	No
Complexity	Highest	High	Medium
Security	High	High	High
Human readable commands	Yes	No	Partially
Firewall-related problems	Yes	No	No
Encryption can be disabled	At any time during the transmission	No	No
Number of commercial server software implementations	Several	Several	Few
Documentation	RFC 4217	IETF Drafts	Implementation's source code
Directory and file system-related features	Yes	Yes	Only file transfers

Table 5.1. Comparison

5.5.2 Conclusion

According to this study, SCP is not an option. It lacks many directory and file system-related features, it is not well documented and it is not supported by many servers. If the CryptoPkg package did not exist it would be the easiest protocol to implement. Nonetheless, it would only be slightly easier to implement than SFTP because SSH would have to be implemented in both cases.

Due to the fact that SFTP is based on SSH and FTPS is based on TLS/SSL, SFTP is less complex than FTPS. In addition, it only needs one connection, avoiding possible firewall-related problems. FTPS is based on FTP, which is an old protocol which needs two connections. SFTP's design is more modern and efficient. These differences are not critical because both protocols offer the same features. Nevertheless, if there were FTPS and SFTP available resources, these differences would be enough to choose SFTP over FTPS.

Feasibility and cost are important parameters and the fact that there exists a UEFI TLS/SSL package which could possibly be ported to the GEP BIOS environment (CryptoPkg package) is critical. Therefore FTPS should be the first choice. Choosing TLS/SSL over SSH has the advantage of being a much more general secure layer which can be used to support possible future applications (i. e. HTTPS-based applications). SSH could also be used to implement many interesting future applications like remote secure terminal login functionalities, but it is not as general as TLS/SSL. Nonetheless, if it is impossible to port the CryptoPkg package to the GEP BIOS environment, SFTP should be chosen.

5.5.3 Future work

The CryptoPkg package should be ported to the GEP BIOS environment if possible. If it is not possible, other possibilities should be explored: writing a SSH library without any support or trying to port SSH code to the GEP BIOS platforms. After TLS/SSL or SSH has been implemented, existing FTPS and SFTP open source implementations should be studied. These open source implementations could be directly ported to the GEP platform's environment. They could also serve as the foundation code for developing a totally new application.

Finally, as shown by [5], nearly every commercial file transfer client software supports several protocols. Therefore, after implementing FTPS or SFTP, giving support to other protocols should be considered. Another option would be to use the underlying secure protocol which has been implemented (TLS/SSL or SSH) and develop other needed secure applications which use that protocol.

Glossary, Bibliography & Appendices

Glossary

AmigaOS It is the default native OS of the Amiga PC family of computers.

API Application Programming Interface. An interface to ease access to software functionalities and facilitate their interaction.

axTLS It is a TLS/SSL project for embedded systems.

BIOS A Basic Input Output System (BIOS) is the lowest-level software used in a computer. Its main function is acting as an interface between the hardware (mainly the chipset and processor) and the operating system. It allows the creation of higher-level operating systems. It also allows the user to control the computer's hardware settings. Booting up the machine and various other system functions are among its responsibilities, as shown by [28].

Boot manager It is the part of the firmware implementation which is responsible for implementing system boot policies.

Bootstrap Protocol It is a network protocol used to deliver IP addresses from configuration servers to clients when a computer is starting.

C Standard library It is a set of headers and library routines used to implement common operations using the ANSI C programming language.

chmod It is a Unix command which lets a user tell the system how much access it should permit to a file or folder.

CIF Component Information File. It is used to describe a component (a driver or an application) in GEP boards' BIOS system.

CP/M Control Program for Microcomputers. It is an OS created by Intel for its microcomputers.

CPU Central Processing Unit. It is the central processor of a computer, where most calculations take place.

CyaSSL It is a TLS/SSL library for embedded systems.

daemon It is a computer program which runs in the background rather than under the direct control of a user.

DHCP Dynamic Host Configuration Protocol. It is a network protocol used to configure hosts in a network before they can communicate with other hosts.

Disk drive It is a machine that reads data from and writes data onto a disk. It has several heads that read and write data on the disk.

Disk storage A general category of storage mechanisms. The data can be digitally recorded by various optical, magnetic, electronic or mechanical methods on the surface of one or several disks.

DXS file Dependency eXpression file. It is used to state the protocol dependencies of a UEFI driver.

EBCDIC Extended Binary Coded Decimal Interchange Code. An 8-bit character encoding used mainly by IBM.

EDK project Also called EDK1 project. It is the open-source component of the Intel Platform Innovation Framework for UEFI.

EDK2 project It is the evolution of the EDK1 project which provides an enhanced build environment.

EEPROM Electrically Erasable Programmable Read-Only Memory. It is a ROM which can be erased and reprogrammed repeatedly using higher than normal electrical voltages.

EFI boot services table A table that contains the firmware entry points for accessing boot services' functions.

EFI runtime services table A table that contains the firmware entry points for accessing runtime services' functions.

EFI Status codes Success, error, and warning codes returned by boot services and runtime services.

EFI system table Also called system table. It is the table that contains the standard input and output handles for an UEFI application and pointers to boot services and runtime services tables.

EFI Toolkit It is a set of tools used to support rapid porting and development of EFI applications.

EFI Extensible Firmware Interface.

FAT16 File Allocation Table 16. It is a disk file system which was first introduced in 1987.

FAT32 File Allocation Table 32. It is a disk file system resulting from the evolution of the FAT16 file system.

Flash memory It is a non-volatile storage medium for electronic devices which can be erased and reprogrammed.

FTP File Transfer Protocol.

FTPS File Transfer Protocol Secure or File Transfer Protocol-SSL. It is a secure protocol which is used to perform file system-based remote operations. It is built using SSL/TLS.

GEP boards Generic Ericsson Processor boards.

GUID Globally Unique Identifier. It is a 128-bit value which is used to differentiate services and structures in the boot services environment.

HII Human Interface Infrastructure. Standard way in which firmware can communicate with an user or an application, defined by UEFI. It is used to communicate firmware-related settings.

HTTPS Hypertext Transfer Protocol Secure. It is the secure version of the HTTP protocol using TLS/SSL.

IBM PC compatible personal computers Personal computers similar to the original IBM PC.

ifconfig It is an application used in Unix-like systems to configure network interfaces which use the TCP/IP protocols.

Image handle It is a handle for a loaded image. They support the UEFI loaded image protocol.

Image's entry point It is the first function executed by a driver or an application.

inetd It is a super-server daemon used to manage Internet services on Unix systems.

Instruction-set Architecture Also called ISA. It includes a specification of the operation codes of a machine and its native commands.

Intel Architecture, 32-bit It is a family of instruction set architectures based on Intel 8086 CPU. It is also called x86 architecture.

Intel Platform Innovation Framework for UEFI It is a platform which describes the primary design elements to construct platform firmware for platforms based on Intel architecture.

Internet Experiment Note Also known as IEN. It is a sequentially numbered document used to publish technical content related to Internet.

IP Internet Protocol.

Itanium It is a 64-bit Intel microprocessors' architecture which was released in 2001.

kernel It is the main component of most OSs. It is used as a bridge between applications and the actual data processing done at the hardware level.

LAN Local Area Network.

Mac OS X It is used to refer to a series of OSs and graphical user interfaces developed by Apple. It is included in all Macintosh computer systems.

Maximum Transmission Unit Also called MTU. It is the size (in bytes) of the largest protocol data unit that a communications protocol layer can pass onwards.

MD5 Message-Digest Algorithm. It is a cryptographic hash function which produces a 128-bit value.

Microcomputers It is a computer with a microprocessor as its CPU.

MS-DOS Also named Microsoft Disk Operating System. It is an OS for computers using the x86 architecture.

MTFTP4 A Multicast TFTP protocol for UEFI platforms. It is described on the UEFI specification [61].

NTFS New Technology File System. It is a disk file system which offers several additional features like encryption mechanisms.

OS An Operating System (OS) is a software product consisting on the combination of programs and data that manages the hardware resources and provides system services used by other applications, as shown by [54].

PC AT Personal Computer Advanced Technology. It is IBM's second-generation PC.

PC Personal Computer. A general-purpose computer which is intended to be operated by an end-user.

PE32+ It is a file format for executables which can be used on x64 platforms.

ping It is an application used to test the connectivity of a host using IP on a network.

PKI Public Key Infrastructure. A system used to create, manage and revoke digital certificates.

Preboot Execution Environments Also known as PXE. It is an environment to boot computers which uses a network interface independently of data storage devices or OSs.

PROM Programmable read-only memory. It is a ROM in which the setting of each bit is controlled by a fuse. It allows to program a ROM after the device is constructed.

RAM disk It is a piece of RAM configured to simulate a disk drive.

RAM Read Access Memory (RAM) is one type of computer memory which provides direct access service to any location (byte).

ROM Read-only memory. It is a storage medium used in electronic devices which cannot be modified or which can be modified slowly or with difficulties. It is mainly used to distribute firmware.

route It is a command used to manipulate a system's routing table.

RSA It is an algorithm for public-key cryptography. RSA stands for Rivest, Shamir and Adleman who first described it.

SCP Secure Copy Protocol. It is a protocol used to transfer files securely between two hosts using SSH.

Secondary storage It is a type of storage which is not directly accessible by the CPU.

SFTP SSH File Transfer Protocol or Secure File Transfer Protocol. It is a secure protocol which is used to perform file system-based remote operations. It is commonly built using SSH.

SHA Secure Hash Algorithm. A cryptographic hash function published by the National Institute of Standards and Technology.

Silicon Disk System It was the first commercially available RAM disk for micro-computers.

SDL file Simple Declarative File. It is a cross-platform language for defining basic data structures.

Sorcerer's Apprentice Syndrome Also known as SAS. It is a network protocol flaw discovered in the early versions of TFTP.

SSH Secure Shell. A network protocol used to transmit data safely over insecure networks.

SSL Secure Sockets Layer. A protocol used to transmit data securely over an insecure network.

super-server daemon It is a type of daemon used on Unix-like systems.

SUSE It is an OS built on top of the open source Linux Kernel which is distributed with software from other open source projects. Its open source version is called openSUSE.

TCP Transmission Control Protocol. It provides ordered, reliable delivery of a stream of bytes from one computer to another.

TLS Transport Layer Security. The evolution of SSL protocol which also provides security to data transfers.

tmpfs It is a temporary file storage facility used by many Unix OSs. It is a file system mounted in volatile memory instead of in a persistent storage device.

Trivial FTP A simpler version of the FTP protocol.

UEFI Boot Services Collection of interfaces and protocols which are present in the boot environment.

UEFI Images Executable files stored in a file system which comply with the UEFI specification.

UEFI Protocol It is the information that defines how to access a certain type of device during boot services.

UEFI Runtime Services They are interfaces which provide access to underlying platform specific hardware which can be useful during OS runtime.

UEFI The Unified Extensible Firmware Interface (UEFI) describes an interface between the OS and the platform firmware, as explained by [61]. It replaces the BIOS firmware interface which was present in all IBM PC-compatible personal computers.

x86-x64 It is an extension of the x86 instruction set which can also be used with 64-bit machines.

Yet another Setup Tool Also called YaST. It is a featured configuration tool from the openSUSE Linux distribution.

Bibliography

- [1] M. Allman, S. Ostermann, *et al.* RFC 2577 - FTP Security Considerations. [Online, ietf.org], May 1999. Available: <http://tools.ietf.org/html/rfc2577> [Accessed 20 September 2011].
- [2] Anonymous. What is R? [Online, r-project.org]. Available: <http://www.r-project.org/> [Accessed 14 November 2011].
- [3] Anonymous. EFI Toolkit Bugs. [Online, sourceforge.net], February 2010. Available: http://sourceforge.net/tracker/?group_id=289014&atid=1223910 [Accessed 23 October 2011].
- [4] Anonymous. BIOS. [Online, wikipedia.org], September 2011. Available: <http://en.wikipedia.org/wiki/BIOS> [Accessed 14 September 2011].
- [5] Anonymous. Comparison of FTP client software. [Online, en.wikipedia.org], October 2011. Available: http://en.wikipedia.org/wiki/Comparison_of_FTP_client_software [Accessed 5 November 2011].
- [6] Anonymous. RAM disk. [Online, en.wikipedia.org], August 2011. Available: http://en.wikipedia.org/wiki/RAM_disk [Accessed 19 September 2011].
- [7] H. Peter Anvin. tftpd - Linux man page. [Online, linux.die.net]. Available: <http://linux.die.net/man/8/tftpd> [Accessed 13 October 2011].
- [8] Abhay Bhushan. RFC 114 - A FILE TRANSFER PROTOCOL. [Online, ietf.org], April 1971. Available: <http://tools.ietf.org/html/rfc114> [Accessed 15 September 2011].
- [9] Bob. List of FTP server software. [Online, en.wikipedia.org], October 2011. Available: http://en.wikipedia.org/wiki/List_of_FTP_server_software [Accessed 5 November 2011].
- [10] Bill Croft and John Gilmore. RFC 951 - BOOTSTRAP PROTOCOL (BOOTP). [Online, ietf.org], September 1985. Available: <http://tools.ietf.org/html/rfc951> [Accessed 12 november 2011].

- [11] Tamara Dean. *Network+ Guide to Networks*. Delmar, 2010.
- [12] R. Droms. RFC 2132 - Dynamic Host Configuration Protocol. [Online, ietf.org], March 1997. Available: <http://tools.ietf.org/html/rfc959> [Accessed 24 October 2011].
- [13] Randy Dunlap. Linux kernel parameters. [Online, cyberciti.biz], November 2005. Available: <http://www.cyberciti.biz/howto/question/static/linux-kernel-parameters.php> [Accessed 15 October 2011].
- [14] A. Emberson. RFC 2090 - TFTP Multicast Option. [Online, ietf.org], February 1997. Available: <http://tools.ietf.org/html/rfc2090> [Accessed 21 October 2011].
- [15] P. Ford-Hutchinson. RFC 4217 - Securing FTP with TLS. [Online, ietf.org], October 2005. Available: <http://tools.ietf.org/html/rfc4217> [Accessed 6 November 2011].
- [16] J. Galbraith. SSH File Transfer Protocol. [Online, filezilla-project.org], July 2006. Available: <http://filezilla-project.org/specs/draft-ietf-secsh-filexfer-13.txt> [Accessed 4 November 2011].
- [17] Intel. EFI Disk Utilities. [Online, intel.dk]. Available: http://www.intel.dk/technology/efi/diskutil_overview.htm [Accessed 15 October 2011].
- [18] Intel. Extensible Firmware Interface (EFI) and Unified EFI (UEFI). [Online, intel.com]. Available: <http://www.intel.com/content/www/us/en/architecture-and-technology/unified-extensible-firmware-interface/efi-homepage-general-technology.html> [Accessed 14 September 2011].
- [19] Intel. Intel Platform Innovation Framework for UEFI. [Online, intel.com]. Available: <http://www.intel.com/technology/Framework/> [Accessed 2 October 2011].
- [20] Intel. EFI Application Toolkit Version 1.10.14.62 Product Release Notes. [Online, sourceforge.net], January 2004. Available: <http://sourceforge.net/projects/efi-toolkit/files/Official> [Accessed 15 October 2011].
- [21] Intel. EFI Developer Kit (EDK) Getting Started Guide. [Online, sourceforge.net], January 2005. Available: <http://sourceforge.net/projects/efidevkit/files/Edk> [Accessed 5 October 2011].
- [22] Intel. EFI Shell Developers Guide. [Online, sourceforge.net], June 2005. Available: <http://sourceforge.net/projects/efi-shell/files/documents/> [Accessed 15 October 2011].

- [23] Intel. EFI Shell User Guide. [Online, sourceforge.net], July 2005. Available: <http://sourceforge.net/projects/efi-shell/files/documents/> [Accessed 1 October 2011].
- [24] Intel. Shell Command Reference Manual. [Online, sourceforge.net], November 2007. Available: <http://sourceforge.net/projects/efi-shell/files/documents/> [Accessed 1 October 2011].
- [25] Intel. EDK 1 Build UEFI Shell and Debug Lab. [Online, feishare.com], April 2009. Available: http://feishare.com/attachments/104_106 [Accessed 13 November 2011].
- [26] Intel. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A. [Online, intel.com], May 2011. Available: <http://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-3a-part-1-manual.html> [Accessed 27 September 2011].
- [27] Tobias Kind. RAMDISK Benchmarks. [Online, fiehnlab.ucdavis.edu], November 2009. Available: <http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/RamDisk/ramdisk-benchmarks.pdf> [Accessed 21 September 2011].
- [28] Charles M. Kozierok. System BIOS. [Online, pcguide.com], April 2001. Available: <http://www.pcguides.com/ref/mbsys/bios/index.htm> [Accessed 14 September 2011].
- [29] J.F. Kurose and K.W. Ross. *Computer Networking*. Pearson Education, 2010.
- [30] Remi Lefebvre and Jean-Pierre Lefebvre. Manual Reference Pages - ATFTPD. [Online, irongeek.com]. Available: <http://www.irongeek.com/i.php?page=backtrack-3-man/atftpd> [Accessed 14 October 2011].
- [31] LibSSH-Team. LibSSH. [Online, libssh.org]. Available: <http://www.libssh.org/> [Accessed 3 November 2011].
- [32] Logic-technology. Traditional BIOS versus UEFI/EFI Framework - an overview. [Online, logic.com], January 2008. Available: <http://www.logic.nl/Products/Technology/BIOS-and-EFI.aspx> [Accessed 14 September 2011].
- [33] G. Malkin and A. Harkin. RFC 2348 - TFTP Blocksize Option. [Online, ietf.org], May 1992. Available: <http://tools.ietf.org/html/rfc2348> [Accessed 20 September 2011].

- [34] G. Malkin and A. Harkin. RFC 2347 - TFTP Option Extension. [Online, ietf.org], May 1998. Available: <http://tools.ietf.org/html/rfc2347> [Accessed 21 September 2011].
- [35] G. Malkin and A. Harkin. RFC 2349 - TFTP Timeout Interval and Transfer Size Options. [Online, ietf.org], May 1998. Available: <http://tools.ietf.org/html/rfc2349> [Accessed 21 September 2011].
- [36] Microsoft. FAT16 vs. FAT32 . [Online, technet.microsoft.com]. Available: <http://technet.microsoft.com/en-us/library/cc940351.aspx> [Accessed 15 October 2011].
- [37] Microsoft. Microsoft Portable Executable and Common Object File Format Specification. [Online, msdn.microsoft.com], September 2010. Available: <http://msdn.microsoft.com/es-es/windows/hardware/gg463125> [Accessed 13 November 2011].
- [38] Daniel Garcia Moral. RAM Disk driver for GEP Boards. Ericsson, November 2011.
- [39] Daniel Garcia Moral. Remote File Access System for GEP Boards: a TFTP UEFI Shell Client and RAM Disk driver. Ericsson, November 2011.
- [40] Daniel Garcia Moral. TFTP UEFI Shell Client for GEP Boards. Ericsson, November 2011.
- [41] NTFS.com. NTFS vs FAT. [Online, ntfs.com]. Available: http://www.ntfs.com/ntfs_vs_fat.htm [Accessed 11 October 2011].
- [42] OpenSSH-Team. OpenSSH. [Online, openssh.org]. Available: <http://www.openssh.org/> [Accessed 2 November 2011].
- [43] OpenSSL-Team. About the OpenSSL Project. [Online, openssl.org]. Available: <http://www.openssl.org/about/> [Accessed 1 November 2011].
- [44] openSUSE Community. openSUSE:Community references. [Online, en.opensuse.org], August 2011. Available: http://en.opensuse.org/openSUSE:Community_references [Accessed 2 October 2011].
- [45] Jan Pechanec. How the SCP protocol works. [Online, blogs.oracle.com], July 2007. Available: http://blogs.oracle.com/janp/entry/how_the_scp_protocol_works [Accessed 6 November 2011].
- [46] Philip. Ramdisk Guide. [Online, speedguide.net], May 2003. Available: <http://www.speedguide.net/articles/ramdisk-guide-131> [Accessed 18 September 2011].

- [47] J. Postel. RFC 764 - TELNET PROTOCOL SPECIFICATION. [Online, ietf.org], June 1980. Available: <http://tools.ietf.org/html/rfc764> [Accessed 21 September 2011].
- [48] J. Postel and J. Reynolds. RFC 959 - FILE TRANSFER PROTOCOL (FTP). [Online, ietf.org], October 1985. Available: <http://tools.ietf.org/html/rfc959> [Accessed 16 September 2011].
- [49] Victor Prijker. Use a RAM disk to speed browsing on slower Macs. [Online, macworld.com], August 2004. Available: <http://hints.macworld.com/article.php?story=20040827132909881> [Accessed 21 September 2011].
- [50] Royce. File system. [Online, en.wikipedia.org], October 2011. Available: http://en.wikipedia.org/wiki/File_system [Accessed 10 October 2011].
- [51] Peter Snyder. tmpfs: A Virtual Memory File System. [Online, solarisinternals.com], 1990. Available: <http://www.solarisinternals.com/si/reading/tmpfs.pdf> [Accessed 27 September 2011].
- [52] K. Sollins. RFC 1350 - THE TFTP PROTOCOL (REVISION 2). [Online, ietf.org], July 1992. Available: <http://tools.ietf.org/html/rfc1350> [Accessed 17 September 2011].
- [53] Karen R. Sollins. IEN 133 - The TFTP Protocol. [Online, ietf.org], January 1980. Available: <http://tools.ietf.org/rfcmarkup?url=ftp://ftp.rfc-editor.org/in-notes/ien/ien133.txt> [Accessed 15 September 2011].
- [54] Stallings. *Operating Systems, Internals and Design Principles*. Pearson, 2005.
- [55] Marco Stornelli. Protected and Persistent RAM Filesystem. [Online, sourceforge.net]. Available: <http://pramfs.sourceforge.net/> [Accessed 15 October 2011].
- [56] Thumperward. Data cluster. [Online, en.wikipedia.org], June 2011. Available: http://en.wikipedia.org/wiki/Data_cluster [Accessed 10 October 2011].
- [57] Tianocore. EFI Toolkit. [Online, sourceforge.net]. Available: http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=EFI_Toolkit [Accessed 5 October 2011].
- [58] Tianocore. Working with the EDK. [Online, sourceforge.net]. Available: <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=Efidevkit> [Accessed 2 October 2011].

- [59] Tianocore. EFI and Framework Open Source Community FAQ. [Online, sourceforge.net], December 2010. Available: <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=FAQ> [Accessed 2 October 2011].
- [60] Tianocore. Nt32 Questions. [Online, sourceforge.net], September 2011. Available: <http://sourceforge.net/apps/mediawiki/tianocore/index.php?title=NT32> [Accessed 13 November 2011].
- [61] Unified-EFI. Unified Extensible Firmware Interface Specification Version 2.1. UEFI, January 2007.
- [62] Unified-EFI. UEFI Shell Specification. UEFI, September 2008.
- [63] Unified-EFI. Unified Extensible Firmware Interface Specification Version 2.3. UEFI, April 2011.
- [64] Alex Verstak. FAT Boot Sector. [Online, averstak.tripod.com], November 1998. Available: <http://averstak.tripod.com/fatdox/bootsec.htm> [Accessed 15 October 2011].
- [65] Andreas Winterer. BIOS from A to Z . [Online, tomshardware.org], January 2006. Available: <http://www.tomshardware.com/reviews/bios-a-z,1200.html> [Accessed 14 September 2011].

Appendices

Appendix A

Block sizes measures' data

Tables A.1 and A.2 show the measures which were performed in order to select the best default block size for the TFTP client application prototype. For each block size, a 10-Megabyte file was downloaded and uploaded ten times. The average throughput was computed from the measures for each block size. The average throughputs are shown in tables A.3 and A.4.

Time to download a 10-megabyte file in milliseconds for different block sizes										
Block size (bytes)	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7	Time 8	Time 9	Time 10
512	7009	6080	6047	6090	6195	6399	6858	6311	6898	6728
7728	581	588	585	573	582	589	578	590	586	581
14944	416	429	428	458	460	422	415	422	449	428
22160	378	373	372	373	374	380	377	378	370	373
29376	286	287	290	290	289	290	286	286	287	286
36592	301	302	303	300	303	300	300	303	303	301
43808	262	260	258	261	260	261	259	262	263	257
51024	255	254	253	253	254	252	252	252	254	255
58240	240	241	241	239	240	242	239	242	240	239
65464	240	241	239	241	239	240	239	240	239	241

Table A.1. Time to download a 10-megabyte file in milliseconds for different block sizes

Time to upload a 10-megabyte file in milliseconds for different block sizes										
Block size (bytes)	Time 1	Time 2	Time 3	Time 4	Time 5	Time 6	Time 7	Time 8	Time 9	Time 10
512	6257	6191	6188	6201	6243	6243	6241	6205	6240	6245
7728	684	688	675	691	684	680	676	688	690	684
14944	430	425	518	517	489	494	454	465	493	474
22160	333	343	339	339	343	343	340	340	339	344
29376	296	296	297	296	298	298	296	295	296	298
36592	282	280	281	280	282	282	281	280	280	280
43808	267	263	265	267	265	263	265	264	263	264
51024	249	250	249	249	249	251	250	249	249	250
58240	240	269	239	238	238	238	240	253	240	249
65464	232	230	230	229	233	230	231	232	232	228

Table A.2. Time to upload a 10-megabyte file in milliseconds for different block sizes

Average download throughput	
Block size (bytes)	Average throughput
512	1.62
7728	17.98
14944	24.23
22160	27.98
29376	36.45
36592	34.77
43808	40.28
51024	41.38
58240	43.64
65464	43.71

Table A.3. Average download throughput in megabytes per second for several block sizes

Average upload throughput	
Block size (bytes)	Average throughput
512	1.68
7728	15.33
14944	22.03
22160	30.81
29376	35.35
36592	37.34
43808	39.63
51024	42.03
58240	42.90
65464	45.45

Table A.4. Average upload throughput in megabytes per second for several block sizes

Appendix B

Throughput measures' data

In order to measure the TFTP client's download throughput and upload throughput 50 measures were taken for each throughput. A 50-megabyte random file was generated and the upload and download times were measured. Then the throughputs were calculated. The results are shown in tables B.1 and B.2.

50 Download throughput measures									
44.06	44.13	44.36	44.43	44.85	44.47	43.95	43.91	44.81	44.28
44.06	44.09	44.13	43.95	44.85	44.10	44.02	43.69	45.12	44.09
44.58	44.89	44.66	43.69	43.87	43.76	44.36	44.73	44.47	44.39
44.58	44.36	44.70	44.85	44.28	44.28	44.93	44.09	43.73	44.70
44.39	44.85	44.58	43.95	44.93	45.04	44.28	44.47	43.98	43.73

Table B.1. 50 Download throughput measures in megabytes per second using default parameters

50 Upload throughput measures									
46.52	45.51	45.20	46.11	44.66	46.11	44.70	45.28	45.87	45.16
45.20	45.16	45.75	45.55	45.08	45.43	46.15	44.62	45.99	45.75
44.77	44.89	44.96	45.24	45.08	44.66	45.28	44.70	45.55	45.12
45.83	45.79	45.75	46.03	45.39	44.93	45.91	45.51	45.63	45.55
46.11	44.62	46.11	45.63	45.59	44.96	44.89	45.55	45.00	45.12

Table B.2. 50 Upload throughput measures in megabytes per second using default parameters

Appendix D

TFTP client functions summary

This appendix is intended to give an explanation about the TFTP client source code functions depicted in Figure 3.7. The code is composed by three source code files as shown in Figure 3.7 whose names are:

- TFTPClient
- tftp
- helpers

The three source code files and all the functions written in them were entirely written by the author of this master thesis project. This source code is included as a component of the UEFI shell's source code which is implemented in the GEP boards. The shell source code is compiled and included in the GEP BIOS binaries which are stored in the GEP BIOS ROM.

The functions written in the source code files make use of other functions:

- UEFI MTFTP4 protocol functions: The UEFI MTFTP4 protocol is described in the UEFI specification [61]. This protocol is implemented in the GEP BIOS code which is stored in the GEP BIOS ROM.
- UEFI boot services: The UEFI boot services are described in the UEFI specification [61]. These services are implemented in the GEP BIOS code which is stored in the GEP BIOS ROM.

The TFTP client functions will be described in the following sections.

D.1 Functions written in the TFTPClient file

D.1.1 TFTPClientMain()

Written by the author of this master thesis project.

D.1.2 TFTPClientGetLineHelp()

Written by the author of this master thesis project. It is invoked when the user uses the *help* UEFI shell's command.

D.2 Functions written in the tftp file

D.2.1 tftpHandle()

Written by the author of this master thesis project.

Boot services which are used by this function

- LocateHandleBuffer()
- CreateChild()
- OpenProtocol()

Helper functions written by the author of this thesis which are used by this function

- UnicodeToAsciiT()

D.2.2 tftpConfigure()

Written by the author of this master thesis project.

D.2.3 tftpLaunch()

Written by the author of this master thesis project.

MTFTP4 functions which are used by this function

- Configure()

D.2.4 tftpGet()

Written by the author of this master thesis project.

MTFTP4 functions which are used by this function

- ReadFile()

Helper functions written by the author of this thesis which are used by this function

- existingFile()
- writeFile()

D.2.5 tftpGetFileSize()

Written by the author of this master thesis project.

MTFTP4 functions which are used by this function

- GetInfo()
- ParseOptions()

D.2.6 tftpPut()

Written by the author of this master thesis project.

MTFTP4 functions which are used by this function

- WriteFile()

Helper functions written by the author of this thesis which are used by this function

- openFile()
- readFile()

D.2.7 tftpCheckPacket()

Written by the author of this master thesis project.

D.3 Functions written in the helpers file

D.3.1 UtoA10()

Written by the author of this master thesis project.

D.3.2 StrToIpT()

Written by the author of this master thesis project.

D.3.3 UnicodeToAsciiT()

Written by the author of this master thesis project.

D.3.4 existingFile()

Written by the author of this master thesis project.

D.3.5 writeFile()

Written by the author of this master thesis project.

D.3.6 readFile()

Written by the author of this master thesis project.

D.3.7 openFile()

Written by the author of this master thesis project.

D.3.8 EfiAsciiStrnCmp()

Written by the author of this master thesis project.

D.3.9 AtoU64()

Written by the author of this master thesis project.

Appendix E

RAM Disk drive functions summary

This appendix is intended to give an explanation about the RAM Disk source code functions and data structures depicted in Figure 4.2. This code is only composed by one source code file called *Ramdisk.c*. This source code file is used to build a GEP BIOS driver component and it is stored in the GEP BIOS ROM. The different types of functions and data structures will be explained in the following sections.

E.1 Data structures

Several data structures are used by the RAM Disk driver. Some of these structures were already written in the EFI Toolkit's source code. One of them was written entirely by the author of this project.

E.1.1 Data structures were already written in the EFI Toolkit's source code

They have been ported to the RAM Disk final prototype and they have not been modified. They are the following:

- RAM Disk device path structure. The instance of this structure in the code is called `RAM_DISK_DEVICE_PATH`.
- RAM Disk device info structure. The instance of this structure in the code is called `RAM_DISK_DEV`.
- FAT16 boot sector structure. The instance of this structure in the code is called `BOOTSEC`.
- FAT16 table structure. The instance of this structure in the code is called `FAT16TABLE`.

E.1.2 Data structure written by the author of this thesis

There was only one data structure written by the author of this thesis. It is the RAM Disk position info structure. The instance of this structure in the code is called `RAM_DISK_POS`.

E.2 Functions

There are two main types of RAM Disk functions: core operation RAM Disk functions and RAM Disk helper functions. The first ones are responsible for the RAM Disk driver operation. The second ones are support functions used by the core operation functions.

Among the core operation functions some of them have been ported without modifications from the EFI Toolkit, others have been ported and then modified and finally there are functions entirely written by the author of this master thesis. Some of the core operation functions which were copied without modifications from the EFI Toolkit are implementations of the `EFI_BLOCK_IO_PROTOCOL` functions which are described in the UEFI specification [61].

There are two types of helper functions: functions allocated in the GEP BIOS libraries, which are not part of the RAM Disk source code and therefore they are not described here and the ones ported without modifications from the EFI Toolkit, which are described here because they are part of the RAM Disk source code.

Finally some UEFI boot services and runtime services are used. Finally some UEFI boot services and runtime services are used. They are not written in the RAM Disk source code. They are used to substitute some Toolkit libraries' helper functions but they are not part of the GEP BIOS libraries. They are part of the GEP BIOS drivers. They are implemented in the GEP BIOS source code and they are stored in the GEP BIOS ROM.

All the functions depicted in Figure 4.2 will be described here. The UEFI boot services, UEFI runtime services and GEP BIOS helper functions used by the core operation functions will only be mentioned if they are relevant for the master thesis project and therefore they have been explained in chapter 4. Nevertheless they will not be described in detail because they are not part of the RAM Disk drive code.

E.2.1 Core operation functions

Functions ported without modifications from the EFI Toolkit which are not `EFI_BLOCK_IO_PROTOCOL` functions' implementations

They are two functions. The first one is the `GetDiskSize()` function. This function was used in the EFI Toolkit version but it is not used in the master thesis' RAM

Disk driver. It is kept in case it is needed in future versions of the system. The second function is the *FormatRamdisk()* function.

Functions ported without modifications from the EFI Toolkit which are EFI_BLOCK_IO_PROTOCOL functions' implementations

They are the following functions:

- *RamDiskReadBlocks()*. It implements the `EFI_BLOCK_READ` function from the `EFI_BLOCK_IO_PROTOCOL`.
- *RamDiskWriteBlocks()*. It implements the `EFI_BLOCK_WRITE` function from the `EFI_BLOCK_IO_PROTOCOL`.
- *RamDiskFlushBlocks()*. It implements the `EFI_BLOCK_FLUSH` function from the `EFI_BLOCK_IO_PROTOCOL`.

Functions ported with modifications from the EFI Toolkit

The only function which was ported with modifications is the *InitializeRamDiskDriver()* function.

It uses the following functions:

- *AllocatePages()*. It is a GEP BIOS helper function.
- *InstallMultipleProtocolInterfaces()*. It is a UEFI boot service.
- *InstallConfigurationTable()*. It is a UEFI boot service.
- *SetVariable()*. It is a UEFI runtime service.

Functions entirely written by the author of the master thesis

They are the following functions:

- *GetSize()*
- *ReportSize()*
- *GetMem()*
- *ReportMem()*

E.2.2 Helper functions

Functions ported from the EFI Toolkit without modifications

They are the following functions:

- CopyMem()
- AllocateZeroPool()
- AllocatePool()
- ZeroMem()
- AtoiU()
- MultU64x32()
- size2spc()

